UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Optimistic Replication in Weakly Connected Resource-Constrained Environments

João Pedro Faria Mendonça Barreto
(Mestre)

Dissertação para obtenção do
Grau de Doutor em Engenharia Informática e de Computadores

Orientador: Doutor Paulo Jorge Pires Ferreira

Júri

Presidente:	Reitor da Universidade Técnica de Lisboa
Vogais:	Doutor Marc Shapiro
	Doutor José Manuel da Costa Alves Marques
	Doutor Paulo Jorge Pires Ferreira
	Doutor José Carlos Alves Pereira Monteiro
	Doutor Carlos Miguel Ferraz Baquero Moreno

Outubro de 2008

Dissertação realizada sob a orientação do

Prof. Paulo Jorge Pires Ferreira
Professor Associado do Departamento de Engenharia Informática
do Instituto Superior Técnico da Universidade Técnica de Lisboa

Preface

The work presented in this dissertation were partially supported, in chronological order, by Microsoft Research; Fundação para a Ciência e a Tecnologia, Ministério da Ciência, Tecnologia e Ensino Superior; and MINEMA. Part of the work was developed in the context of the Haddock-FS (Microsoft Research) and UbiRep (FCT SFRH/BD/13859) projects.

The following peer-reviewed publications, technical reports and invited talks partially describe the work and results in the dissertation.

International Journals

1. João Barreto and Paulo Ferreira. Version Vector Weighted Voting Protocol: efficient and fault-tolerant commitment for weakly connected replicas. *Concurrency And Computation: Practice And Experience*, Volume 19, Issue 17 , Pages 2271 – 2283, December 2007. John Wiley & Sons, Ltd.

International Conferences and Workshops (with Peer-Reviewing)

1. João Barreto and Paulo Ferreira. The Obscure Nature of Epidemic Quorum Systems. *Proceedings of ACM HotMobile 2008: The Ninth Workshop on Mobile Computing Systems and Applications*, Napa Valley, CA, USA, February 2008. ACM Press.

2. Pierre Sutra, João Barreto and Marc Shapiro. Decentralised Commitment for Optimistic Semantic Replication. In *Proceedings of the 15th International Conference on Cooperative Information Systems (CoopIS 2007)*, Vilamoura, Portugal, November 2007. Springer Verlag Lecture Notes in Computer Science (LNCS).

3. João Barreto, Paulo Ferreira and Marc Shapiro. Exploiting our computational surroundings for better mobile collaboration In *Proceedings of the 8th International Conference on Mobile Data Management (MDM'07)*, pp. 110-117, Mannheim, Germany, May 2007. IEEE.

4. João Barreto and Paulo Ferreira. An Efficient and Fault-Tolerant Update Commitment Protocol for Weakly Connected Replicas. In *Proceedings of Euro-Par 2005*, pp. 1059-1068, Lisboa, August 2005. Springer Verlag Lecture Notes in Computer Science (LNCS).

5. João Barreto and Paulo Ferreira. A Highly Available Replicated File System for Resource-Constrained Windows CE .Net Devices. In *Proceedings of the 3rd International Conference on .NET Technologies*, Pilsen (Czech Republic), May 2005.

Other Communications in International Conferences

1. João Barreto and Paulo Ferreira. Understanding Epidemic Quorum Systems. *Poster Session at the 2nd Eurosys*, Lisbon, March 2007.

2. João Barreto and Paulo Ferreira. Efficient File Storage Using Content-based Indexing. *Poster Session at the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton (UK), October 2005.

(Selected) Technical Reports

1. João Barreto and Paulo Ferreira. Efficient Corruption-Free Duplicate Elimination in Distributed Storage Systems. *INESC-ID Technical Report 49/2008*, September 2008.

2. João Barreto and Paulo Ferreira. The Availability and Performance of Epidemic Quorum Algorithms. *INESC-ID Technical Report 10/2007*, February 2007.

Invited Talks

1. *Decoupled Eventual Consistency*. Seminar at the Distributed Systems Laboratory of the Swedish Institute of Computer Science, September 2006.

2. *Decoupling Agreement from Propagation of Updates in Optimistic Replication*. Seminar at LIP6 - Univ. Paris VI (Pierre et Marie Currie), Paris, March 2006.

3. *Haddock-FS: Replicação Optimista em Ambientes Móveis Colaborativos*. Seminar at Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, November 2005.

4. *Enhancing Consistency in Optimistically Replicated Collaborative Scenarios*. Seminar at LPD - École Polytechnique Fédérale de Lausanne, April 2005.

5. *Optimistic Consistency with Version Vector Weighted Voting*. 2nd MINEMA Workshop, Lancaster, December 2004.

Publications Out of the Scope of the Thesis

The following publication has been produced during the doctoral programme. However, its subject lies out of the scope of the present dissertation.

1. Sebastien Baehni, João Barreto, Patrick Eugster, Rachid Guerraoui. Efficient Distributed Subtyping Tests. *Inaugural International Conference on Distributed Event-based Systems (DEBS 2007)*, Toronto (Canada), June 2007. ACM International Conference Proceeding Series.

Resumo

A presente dissertação tem como objectivo principal melhorar a produtividade de actividades colaborativas suportadas por computadores, através do suporte à partilha de dados em ambientes de rede de fraca qualidade (tais como a Internet ou ambientes de computação móvel e ubíqua). Para assegurar o anterior objectivo, baseamo-nos em replicação de dados optimista, uma técnica bem estudada. No entanto, as limitações ao nível das ligações de rede dos ambientes considerados, assim como as limitações dos equipamentos usados em alguns desses ambientes, são obstáculos cruciais a que replicação optimista seja de facto útil e eficaz.

Esta dissertação propõe inovações às técnicas clássicas de replicação optimista, que conseguem (i) mais rapidamente assegurar consistência forte, (ii) com o custo de menos trabalho cancelado, ao mesmo tempo que (iii) minimiza tanto o volume de dados trocado pela rede como o armazenado nas réplicas.

Primeiramente, concebemos um protocolo descentralizado de consistência baseado em votação epidémica e vectores de versão. O protocolo oferece melhor tolerância a faltas que protocolos de consistência centralizados, ao mesmo tempo que assegura um desempenho comparável aos últimos. Complementamos o protocolo com novos resultados que permitem uma melhor compreensão de protocolos de votação epidémicos, entre os quais o protocolo que propomos se enquadra.

Em segundo lugar, descrevemos como melhorar protocolos de consistência genéricos por exploração de outros recursos computacionais co-existentes com o sistema replicado, recursos esses que são normalmente ignorados por sistemas replicados tradicionais. Propomos o uso desses recursos, nomeadamente nós fixos ou móveis, como transportadores de pacotes de meta-dados de consistência, de pequena dimensão. O transporte desses pacotes entre nós do sistema replicado traz, em situações de ligação fraca do sistema, ganhos nos três eixos supracitados (i, ii e iii).

Em terceiro lugar, concebemos uma solução para sincronização e armazenamento eficiente de réplicas em sistemas com escritas baseadas em valor. A solução baseia-se na informação de controlo de versões mantida pelo sistema replicado para eliminar, de forma correcta, dados redundantes entre versões e/ou entre objectos. A solução elimina desvantagens fundamentais da abordagem de comparação de valores de dispersão, proposta em trabalhos recentes de grande relevância.

Foram implementados protótipos de todas as contribuições acima mencionadas, tendo estas sido avaliadas tanto com cargas simuladas como com cargas reais. Os

resultados obtidos mostram que, quando comparadas com o estado da arte, as nossas propostas alcançam ganhos importantes em todos os três eixos anteriores. É de sublinhar que os benefícios das nossas contribuições tornam-se mais substanciais à medida que as condições experimentais se aproximam das características das redes e dos dispositivos que são habitualmente encontrados em ambientes de computação móvel e ubíqua.

Palavras chave: Replicação Optimista, Redes Móveis, Computação Ubíqua, Protocolos de Consistência, Algoritmos Epidémicos, Detecção Distribuída de Redundância de Dados, Sistemas de Ficheiros Distribuídos.

Abstract

This thesis aims at improving the productivity of collaborative users by supporting highly available data sharing in poorly connected environments (e.g. Internet, ubiquitous and pervasive computing environments). We use optimistic replication, a well known technique to attain such a goal. However, the poor connectivity of such environments and the resource limitations of the equipments used are crucial obstacles to useful and effective optimistic replication.

We propose novel improvements to optimistic replication that allow (i) faster strong consistency (ii) with less aborted work, while (iii) minimizing both the amount of data exchanged between and stored at replicas.

First, we introduce a decentralized update commitment protocol based on epidemic weighted voting and version vectors. It provides better fault-tolerance than primary commit schemes while ensuring comparable performance. We complement such a protocol with novel results that help our understanding of epidemic quorum protocols, such as ours.

Second, we describe how to generically improve update commitment by a novel exploitation of other computational resources, co-existent with the replicated system, whose presence replicated systems normally neglect. We use such resources, namely other stationary or mobile nodes, as carriers of lightweight consistency meta-data that help to improve the replicated system in the above three dimensions (i, ii and iii).

Third, we devise an approach for state-based replica synchronization and storage that relies on version tracking information to achieve efficient and corruption-free data deduplication. It eliminates crucial shortcomings of the prominent compare-by-hash deduplication approach.

We have implemented prototypes of our contributions and evaluated them, both with simulated and real-world workloads. Our results show that, when compared to related state-of-the-art solutions, our proposals attain relevant improvements in all the previous three dimensions. Most importantly, the benefits of our contributions are more substantial as experimental conditions move closer to the network and device characteristics that are commonplace in ubiquitous and pervasive computing environments.

Keywords: Optimistic Replication, Mobile Networks, Ubiquitous Computing, Commitment Protocols, Epidemic Algorithms, Data Deduplication, Distributed

File Systems.

Acknowledgements

I would like to thank Prof. Paulo Ferreira, my Ph.D. advisor, for the energetic and omnipresent support and motivation all through the Ph.D. programme, and especially during the writing of this dissertation. His influence was fundamental for my (good) option to enroll in the Ph.D. programme and to pursue an academic research career, and I am most grateful for that.

I would also like to acknowledge the members of the Comissão de Acompanhamento de Tese, namely Prof. Paulo Ferreira, Prof. Carlos Baquero, Prof. José Alves Marques and Prof. José Carlos Monteiro, for the time spent reading a preliminary version of this dissertation and for the very valuable comments and suggestions given at the meeting held in April 2007.

To Fundação para a Ciência e a Tecnologia, Ministério da Ciência, Tecnologia e Ensino Superior, MINEMA and Microsoft Research for their financial support.

To Dr. Marc Shapiro, Pierre Sutra and the members of the Regal Team at Université Paris VI, for hosting a memorable internship in Paris and introducing me to the challenging and interesting world of semantic optimistic replication.

To Rachid Guerraoui, for the two very inspiring research visits to LPD, EPFL, in beautiful Lausanne, where a substantial portion of the present document was written. To him and to the other LPD members, thanks for the great discussions and new ideas (waiting to be carried on) on new topics, football matches and skiing.

To my office mate, Luís Veiga, for his insightful advices and help on the often Dantesque tasks of researching and teaching.

To Pedro Gama, my twin soul as a Ph.D. student, for his incitement to finish this dissertation, and who I owe a lunch.

To Diogo Paulo and João Paiva, for their helpful reviews of the document.

To all other past and present members of the Distributed Systems Group at Inesc-ID in Lisboa with whom I have closely shared the last six years, André Zúquete, Edgar Marques, Carlos Ribeiro, João Garcia, João Leitão, João Nuno Silva, José Mocito, Liliana Rosa, Luís Rodrigues, Nuno Carvalho, Nuno Santos, Paolo Romano, Pedro Fonseca, Ricardo Lebre, Rodrigo Rodrigues, Rui Joaquim. Thanks for the refreshing discussions, not just on distributed systems, but especially those out of that topic.

To my family (needless to say), for their immense support.

To my friends, who regularly share the special and weird magic of Lisboa with me. To Luisa.

Contents

Chapter 1

Introduction

Consider a team of co-workers that wishes to write a report in a collaborative fashion. For such a purpose, they create a document file, replicate it across the (possibly mobile) personal computers each colleague holds, and occasionally synchronize replicas to ensure consistency. Using a text editor, each worker is then able to read and modify her replica, adding her contributions asynchronously with the remaining colleagues. One may envision interesting scenarios of productive collaboration. A user may modify the report even while out of her office, replicated in a laptop or hand-held computer she carries while disconnected from the corporate wired network. Further, such a worker may meet other colleagues carrying their laptops with report replicas and, in an ad-hoc fashion, establish a short term work group to collaboratively work on the report.

Besides the shared document editing application, one may consider a wide range of other applications and systems; examples include asynchronous groupware applications [Wil91, CS99] such as cooperative engineering or software development [C$^+$93, CMP04, Cho06], collaborative wikis [LC01, IOM$^+$07], shared project and time management [LKBH$^+$88, Byr99], and distributed file [Now89, MSC$^+$86] or database systems [TTC$^+$90].

The previous example illustrates the potential collaborative scenarios that the emerging environments of ubiquitous and pervasive computing [Wei91, FZ94] allow. One may even conceive more extreme scenarios, where the fixed network infrastructure is even less likely to be present. For example, data acquisition in field work, emergency search-and-rescue operations, or war scenarios [RT99]. In all these scenarios, collaboration through data sharing is often crucial to the activities the teams on the field carry out.

Optimistic replication is especially interesting in all previous scenarios, due to their inherent weak connectivity. They are based on mobile networks, whose bandwidth is lower than in local-area wired networks, and where network partitions and intermittent links are frequent. Further, mobile nodes reduce their up-time due to battery limitations. Moreover, a fixed network infrastructure, such as a corporate local area network, may not always compensate for the limitations of mobile

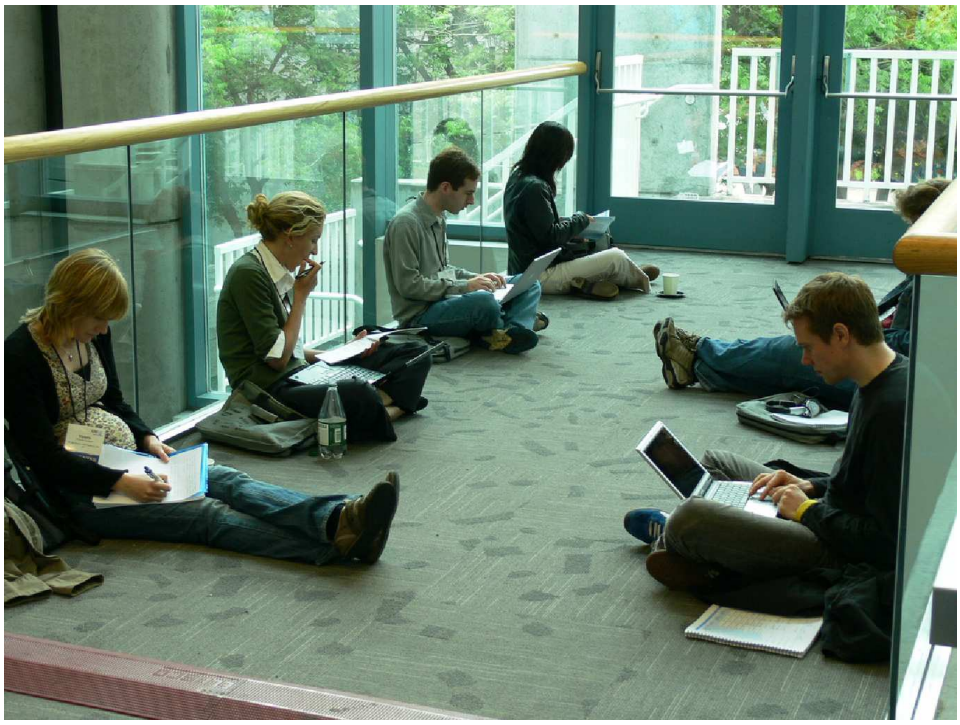


Figure 1.1: Example of a weakly connected and resource-constrained network environment. One may envision interesting scenarios of productive collaboration if effective data sharing is made possible among the mobile users.

networks. In fact, access to such an infrastructure is often supported by wireless connections such as IEEE 802.11 [IEE97], UMTS [3GP] or GPRS [AB98]; these are typically costly in terms of price and battery consumption, and are seldom poor or intermittent due to low signal. Hence, access to the fixed infrastructure is often minimized to occasional connections. Furthermore, access to the fixed network infrastructure is often established along a path on a wide-area network, such as the Internet; these remain slow and unreliable [ZPS00, DCGN03].

Optimistic replication, in contrast to traditional replication (i.e., pessimistic replication), enables access to a replica without *a priori* synchronization with the other replicas. Hence, it offers highly available access to replicas in spite of the above limitations (among other advantages over traditional replication [SS05]). As collaboration in ubiquitous and pervasive computing environments becomes popular, the importance of optimistic replication increases.

Inevitably, however, consistency in optimistic replication is challenging [FB99, YV01, Ped01]. Since one may update a replica at any time and under any circumstance, the work of some user or application at some replica may conflict with concurrent work at other replicas. Hence, consistency is not immediately ensured. A *replication protocol* is responsible for disseminating updates among replicas and

eventually scheduling them consistently at each of them, according to some consistency criterion. The last step of such a process is called *update commitment*. Possibly, it may involve rolling back or aborting updates, in order to resolve conflicts.

The usefulness of optimistic replication depends strongly on the effectiveness of update commitment along two dimensions:

- Firstly, update commitment should arrive rapidly, in order to reduce the frequency by which users and application are forced to access possibly inconsistent data.
- Secondly, update commitment should be achieved with a minimal number of aborted updates, so as to minimize lost work.

Orthogonally, such requirements must be accomplished even in spite of the limitations of the very environments that motivate optimistic replication. Namely:

- The relatively low bandwidth of mobile network links, along with the constrained energy of mobile devices, requires an efficient usage of the network.
- Similarly, bandwidth of wired wide-area networks is a limited resource; hence, when such networks are accessible, their bandwidth must be used wisely by the replication protocol.
- Further, node failures and network partitions have non-negligible probability, both in wired wide-area networks and, especially, in mobile networks; their implications are crucial to any distributed system.
- Finally, the memory resources of mobile devices are clearly constrained when compared to desktop computers; the storage overhead of optimistic replication must be low enough to cope with such limitations.

Although much research has focused on optimistic replication, existing solutions fail to acceptably fulfill the above two requirements (rapid update commitment and abort minimization). As we explain later in the thesis, proposed optimistic replication solutions either do not support update commitment, or impose long update commitment delays in the presence of node failures, poor connectivity, or network partitions. Some commitment approaches are oblivious to any application semantics that may be available; hence, they adopt a conservative update commitment approach that aborts more updates than necessary. Alternatively, semantic-aware update commitment is a complex problem, for which semantically-rich fault-tolerant solutions have not yet been proposed. Finally, more intricate commitment protocols that aim at fulfilling the above requirements have significant network and memory overheads, unacceptable for environments where such resources are scarce.

Thus, the goal of this thesis is twofold: i) to improve the speed at which updates become committed; and ii) to achieve that at the cost of a minimal number of aborted updates, relying on the available semantic information. We must attain this goal while minimizing both the amount of data exchanged between replicas, and the amount of memory being used. These requirements are fundamental in weakly connected and resource-constrained environments such as ubiquitous and pervasive computing ones.

1.1 Contributions

To achieve the goal mentioned above, while respecting the requirements, this thesis presents a solution along the following lines: i) voting-based commitment protocols allowing rapid commitment even in the presence of failures or partitions, ii) decoupling commitment agreement from update propagation for improved update commitment by exploitation of nodes surrounding the replicated system, and iii) appropriate application of content-based indexing for efficient update storage and propagation. A more detailed description of our contributions follows:

1. Novel Results on Epidemic Quorum Systems for Update Commitment [BF08b, BF07a, BF07c].

   We formalize epidemic quorum systems and provide a novel, generic characterization of their availability and performance. Our contribution highlights previously undocumented trade-offs and allows an analytical comparison of proposed epidemic quorum systems [Kel99, HSAA03].

2. Version Vector Weighted Voting Protocol (VVWV) [BF07b, BF05b, SBS07].

   We propose a novel commitment protocol, devised for systems where the happens-before relation [Lam78] between updates is the only available source of semantic information. It is an epidemic weighted voting protocol based on version vectors [PPR+83]. The use of a voting approach eliminates the single point of failure of primary commit approaches [PST+97]. Moreover, VVWV introduces a significant improvement over basic epidemic weighted voting solutions [Kel99] by being able to commit multiple updates quicker, in a single election round. VVWV was later extended to exploit a rich semantic repertoire [SBK04, SB04], in joint work with Pierre Sutra and Marc Shapiro [SBS07].

3. Decoupled Commitment by Consistency Packet Exchange [BFS07].

   We propose the decoupling of the commitment agreement phase from the remaining phases, notably update propagation, in generic optimistic replication protocols. We show that it reduces commitment delay and aborted updates, and allows more network-efficient propagation of updates. Further, we propose an application of the same principle in environments of high

topological dynamism and frequent partitioning, e.g. mobile networks, or ubiquitous, or pervasive computing environments. With decoupled commitment agreement, extra nodes surrounding the group of replica nodes may be temporary carriers of lightweight consistency meta-data. We show that their exploitation may accelerate commitment, avoid aborts, and allow more network-efficient update propagation.

4. Versioning-Based Deduplication [BF05a, BF05c, BF08a].

   We address the problem of adapting state-transfer replicated systems to environments with bandwidth and memory constraints.

   We propose a novel approach for efficient replica storage and synchronization when state-based updates are employed. Our contribution exploits cross-version and cross-object similarities. Its novelty lies in the combination of content-based indexing [MCM01, TM98] with the version state that the replication protocol already maintains. When compared with other existing approaches, our contribution eliminates important drawbacks. Namely, it relies on deterministic similarity detection, instead of data-corruption-prone compare-by-hash techniques; it allows faster distributed similarity detection than compare-by-hash; and it is designed for efficient storage and local data access.

The following table sums up the benefits of each contribution, mapping them against three main improvement dimensions.

Contribution	Faster update commitment	Fewer aborts	Smaller space and network requirem.
Results on Epidemic Quorum Systems	X		
VVWV Protocol	X		
Consistency Packet Exchange	X	X	X
Versioning-Based Deduplication			X

Table 1.1: Direct improvements of each contribution on each axis.

We implemented two prototypes incorporating the contributions: a simulator of distributed environments with poor and partitioned connectivity, and a distributed replicated file system. Such prototypes allowed us to thoroughly evaluate our contributions against network settings and application workloads that are close to the real environments that we target.

In conclusion, the high-level overall goal of this thesis is to improve users' productivity by supporting data sharing with high availability and performance. Optimistic replication is a well known technique to attain such a goal but its usefulness strongly depends on the underlying commitment protocol's ability to ensure fast and reliable replica consistency. This thesis presents new algorithms, protocols, a system architecture, a simulator and a prototype for real deployment. Overall, the proposed system ensures rapid update commitment at the cost of low aborted updates, even in spite of poor connectivity or limited device memory.

1.2 Document Outline

The remainder of the document is organized as follows. Chapter 2 provides the background for the document by describing related state-of-the-art work. Chapter 3 introduces the proposed architecture, as well as the system model and terminology followed in this document.

Chapter 4 describes the commitment protocol. Chapter 5 proposes its extension in order to exploit computing resources surrounding the replicated system. Chapter 6 addresses update propagation and replica storage issues, proposing an architecture for efficient replica storage and synchronization when state-based updates are employed.

Chapter 7 describes the implementation of the proposed contributions, while Chapter 8 evaluates them. Finally, Chapter 9 draws conclusions and directions for future work.

Chapter 2

Related Work on Optimistic Replication

A replicated system maintains replicas of a set of logical objects at distributed machines, called *sites* (also called replica managers or servers). A logical object can, for instance, be a database item, a file or a Java object. Most replicated systems intend to support some collaborative task that a group of users carries on [SS05, Bar03, DGMS85].

It is the ultimate objective of a replicated system to ensure that distributed replicas are consistent, according to some criterion that the users of the system expect. A number of projects try to achieve the latter objective by the optimistic approach for data replication. The reasons include increased availability, performance, and/or scalability [SS05].

In this section, we study relevant design choices that have been proposed for each essential aspect of optimistic replication and survey the state-of-the-art optimistic replication solutions.

The remainder of the section is organized as follows. Section 2.1 starts by addressing the trade-off between consistency and availability in data replication, describing a number of relevant consistency guarantees. Section 2.2 then describes traditional, pessimistic replication, exposing its main limitations. Section 2.3 then introduces optimistic replication.

We then analyze in greater detail some central design choices in optimistic replication. Section 2.4 addresses partial and full replication. Section 2.5 discusses the important role of mechanisms for tracking the happens-before relationship among updates and versions, describing relevant solutions. Section 2.6 then analyzes approaches for update scheduling and commitment. We then address schemes for efficient replica storage and synchronization in Section 2.7, and present additional complementary adaptation techniques in Section 2.8.

Having discussed the main design choices, their inter-dependencies, strengths and weaknesses, we present examples of relevant concrete systems that rely on optimistic replication in Section 2.9. Finally, Section 2.10 summarizes the overall

characteristics of the previous systems and draws conclusions.

2.1 Consistency Guarantees

When one considers centralized access to data objects of any type, an implicit notion of correctness is usually well known and accepted by users and applications. For instance, database systems generally offer the properties of atomicity and serializability as essential components of their correctness guarantees [DGMS85]. Moving on to a different context, most file systems supply familiar file sharing semantics such as Unix semantics [LS90].

In a replicated system, however, a logical object may be replicated in various physical replicas within the network. In this case, ensuring that each physical replica is individually correct according to the above properties is not sufficient. Instead, we must also consider correctness invariants that must be satisfied amongst all physical replicas of a logical object. Such invariants define the notion of *consistency* in the replicated system.[1]

Consider, for instance, that a logical object containing information concerning a bank account is replicated in two physical replicas, located at distinct machines. Initially, the bank account had a balance of 100 in both replicas. Now assume that the physical replicas become separated by a network partition. If both of these replicas are allowed to be updated while the network is partitioned, an incorrect replica state may be reached. For example, if a withdraw request of 60 is performed at one of the partitions and then a withdrawal of 70 is also accepted at the other partition. Since the replicas are unable to contact each other, the updates issued at each one are not propagated between them. Hence, an inconsistent state is reached, since the logical bank account can be seen as having different balances depending on which replica the read request was made on. Moreover, the bank account semantics were violated, since a total withdrawal of 130 was, in fact, allowed when the account had insufficient funds for it.

One possible solution for ensuring that the bank account semantics are verified in the previous example would be to prohibit any write or read request to the physical replicas in the presence of network partitions. Conversely, the same behavior could be followed in the presence of a replica's machine failure. This would be a pessimistic strategy, which prevents inconsistencies between replicas by decreasing their availability. As a result, strong consistency guarantees would be ensured, which would fulfill the correctness criteria of most applications.

When network partitioning or server failure have non negligible probabilities, availability and consistency become two competing goals that a replicated system must try to achieve to some extent [YV00]. The trade-off between these contending vectors can yield replicated solutions which offer weaker consistency guarantees as a cost for improved availability. Those solutions are called optimistic.

[1] Consistency can also be considered amongst different objects. For simplicity, we will restrict our discussion to single-object granularity.

One important aspect to emphasize is that different applications can have different correctness criteria regarding the replicated logical objects they access. Therefore, weaker consistency guarantees provided by optimistic strategies may still meet the needs of some application semantics.

For instance, some applications may consider it acceptable for replicas to temporarily have distinct values, provided that they eventually reach a correct and consistent value. In particular, if a bank account was allowed to have a negative balance, the above example might be acceptable according to such a correctness criteria. In spite of having distinct balance values during the partition, the physical replicas would propagate each one's updates upon being reconnected. In the end, both physical replicas would have the same consistent balance value, -30.

The next subsections present some consistency guarantees that replicated systems normally offer. We describe all of them with reference to a general situation where some distributed clients request operations upon some logical object, each client accessing its local replica. Requests may either be for read or write operations. Each replica, a, thus executes an ordered sequence of operations, $o_{a,1}, o_{a,2}, o_{a,3}, ..$, that it accepted from requests of local clients. In the case of write requests, the local replica conceptually represents their effects by an update, denoted $u_1, u_2, u_3, ..$, which it may propagate to other replicas. We assume that operations requests are synchronous, i.e., a client waits for the last operation requested to complete before requesting the next one.

2.1.1 Strong Consistency Guarantees

In a replicated system, one can consider a virtual ordering of the clients' operations which reflects one possible correct execution if a centralized system was used. The strong consistency guarantees that we describe ahead take one such possible ordering as a reference, which the actual orderings that each replica executes should be consistent with.

Enforcing strong consistency guarantees requires adopting pessimistic strategies for replication, which we discuss in Section 2.2.

2.1.1.1 Linearizability

The most strict consistency guarantee is *linearizability* [GCK01]. A replicated object is said to be linearizable if, for any series of operations performed upon it by any client, there is some canonical ordering of those operations such that:

(1) The interleaved sequence of operations meets the specification of a single correct copy of the objects.

(2) The order of operations in the canonical ordering is consistent with the *real times* at which the operations occurred in the actual execution.

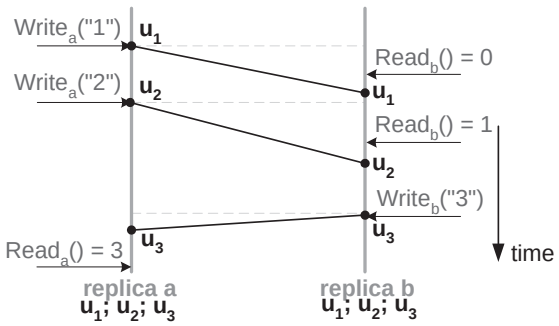


Figure 2.1: An example of an execution that is sequentially consistent, with respect to the canonical ordering: $Read_B() = 0$, $Write_A$("1"), $Read_B() = 1$, $Write_A$("2"), $Write_B$("3"), $Read_A() = 3$.

Though desirable in an ideal replicated system, such a real-time requirement is normally impractical under most circumstances [GCK01].

2.1.1.2 Sequential Consistency

A less strict consistency criterium is called *sequential consistency*. Basically, it discards the real-time requirement of linearizability and only looks at the order of operation requests performed at each client, the *program order*. More specifically, the second condition of the linearizability criterium is now changed to:

(2) The order of operations in the canonical ordering is consistent with the program order in which individual clients executed them.

Figure 2.1 depicts an execution of a 2-replica system that is sequentially consistent. Consider the following canonical ordering: $Read_B() = 0$, $Write_A$("1"), $Read_B() = 1$, $Write_A$("2"), $Write_B$("3"), $Read_A() = 3$. Clearly, the order by which each replica (*a* and *b*) executes its local operations is consistent with the canonical ordering. Notice that, in contrast, the above execution is not linearizable, since the values returned by both $Read_B$ operations are not consistent with the real times at which both $Write_A$ operations occur.

The definition of sequential consistency closely resembles the transactional property of serializability [Pap79, BG83]. However, sequential consistency, just as every replication consistency guarantee that we describe herein, does not include any notion of operation aggregation into transactions. For this reason, sequential consistency and serializability should be regarded as distinct concepts.

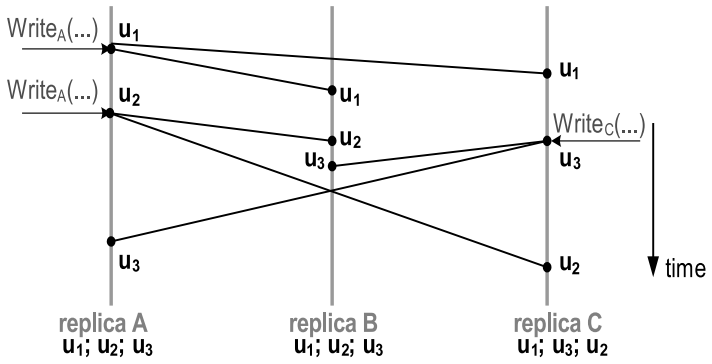


Figure 2.2: An example of FIFO ordering of updates.

2.1.2 Relaxed Consistency Guarantees

Relaxed consistency guarantees are normally specified in terms of the properties that are ensured in the way updates are ordered at each replica. In contrast to strong consistency guarantees, the notion of a canonical operation ordering is no longer used as a reference by relaxed consistency guarantees.

The inherently weak requirements of relaxed consistency guarantees enable the use of optimistic replication strategies the achieve them.

2.1.2.1 FIFO Ordering

One weak consistency guarantee is *FIFO* ordering. It considers the partial ordering between the operations requested by each individual client. It guarantees that such partial orderings are preserved by every updates orderings applied to the replicas.

We illustrate FIFO ordering in Figure 2.2, in a system with three replicas, *a*, *b* and *c*. Clearly, the partial order between operations requested by each individual client is ensured at every replica. Notice that, since FIFO ordering is not total, not every replica applies the updates in the same sequence; hence, it does not guarantee convergence to a common value.

2.1.2.2 Happens-before Ordering

A stronger consistency guarantee considers the *happens-before relationship*, which Lamport [Lam78] initially introduced for generic events. In the particular case of operations issued in replicated system, we may simply define the happens-before relationship between updates as follows:

If an update u_x is issued upon the value of some replica, which results from the execution of a sequence of operations $u_1, u_2, .., u_n$, then each operation $u_1, u_2, .., u_n$

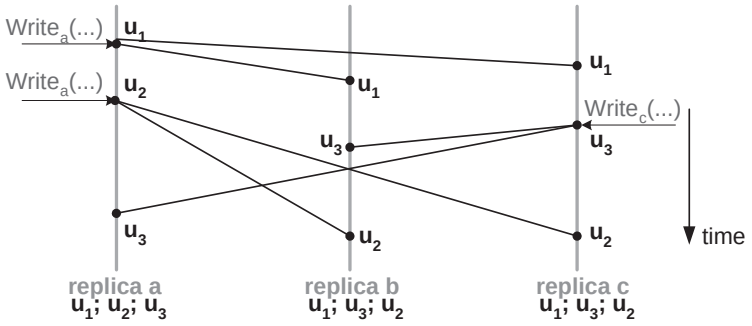


Figure 2.3: An example of happens-before ordering of updates.

happens-before u_x.

Each operation $u_1, u_2, .., u_n$ above was either issued at the local replica, or issued at some other replica and then it was propagated to the first replica. Hence, it is easy to show that the above definition is a simple corollary of Lamport's original definition of happens-before between events in a distributed system [Lam78].

A system guarantees happens-before ordering if and only if the operation ordering applied at each replica satisfies the happens-before partial order among the interleaved updates. It follows directly from the above definition that, since earlier updates requested by some client happen-before later updates that the same client issued, happens-before ordering is stronger than FIFO ordering.

Figure 2.3 depicts an example execution that satisfies happens-before ordering. Clearly, the update ordering at every replica ensures the happens-before relationships between updates ($u_1 \rightarrow u_2$ and $u_1 \rightarrow u_3$). Similarly to FIFO ordering, happens-before ordering is not total, hence it does not ensure replica convergence.

The happens-before ordering is commonly called causal ordering, which is misleading. While the happens-before relationship *captures* the causality relationship between events, it is not the same relationship. As an example, consider that the owner of the bank account in the previous example received a cheque of 100 and, consequently, some client issues a credit operation of 100 upon the local replica. The updates that the local replica had already applied (such as the withdrawals from the previous example) before the credit operation happens-before the latter; nevertheless, crediting the cheque would have happened no matter which updates had executed before at the local replica. In other words, the previous updates happen-before the credit operation but do not causally precede it.

2.1.2.3 Total ordering

A system offers total ordering if the operation ordering at every replica respects a total ordering established between all updates issued in the system.

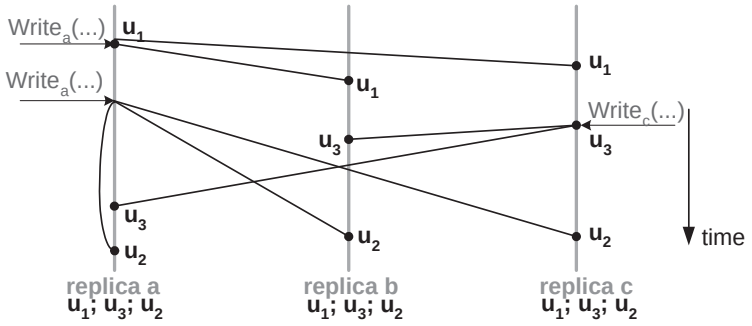


Figure 2.4: An example of total ordering of updates.

One important property of a system offering total ordering is that it guarantees that, once two distinct replicas of some object learn of the same operation requests, their physical values will be identical. This is not true with the other weak consistency guarantees described previously, which only enforce partial orderings.

Figure 2.4 illustrates total ordering of updates. In the example, the total order $u_1 < u_3 < u_2$ is ensured at every replica.

Weak consistency criteria can be combined in order to provide stronger consistency guarantees. One common consistency criterion is obtained by ensuring both total and happens-before orderings.

2.1.2.4 Strict happens-before ordering

Some systems require a stronger variant of happens-before ordering that eliminates the possibility of concurrent updates being executed at the same replica. Strict happens-before ordering offers such a guarantee.

A replicated system offers strict happens-before ordering if and only if: given an update u_i, for every update u_k preceding u_i at the ordering considered by a given replica, we have u_k happens-before u_i.

Contrary to the other weak consistency guarantees described so far, this consistency guarantee prohibits each individual replica of considering concurrent updates in the same ordering. Instead, in the presence of multiple concurrent updates, a replica must accept only one of them and discard the others.

We illustrate strict happens-before ordering by revisiting the previous example, of happens-before ordering (in Figure 2.3), and imposing the former guarantee. Figure 2.5 depicts the resulting execution. In order to ensure strict happens-before ordering, replicas have to discard certain concurrent updates (since u_2 and u_3 are concurrent).

An important property of replicated systems that provide strict causal ordering is that each update is only applied in the context in which some client issued the corresponding write operation. Let us consider that some client writes to a replica when its value results from some sequence of previous updates. Strict happens-

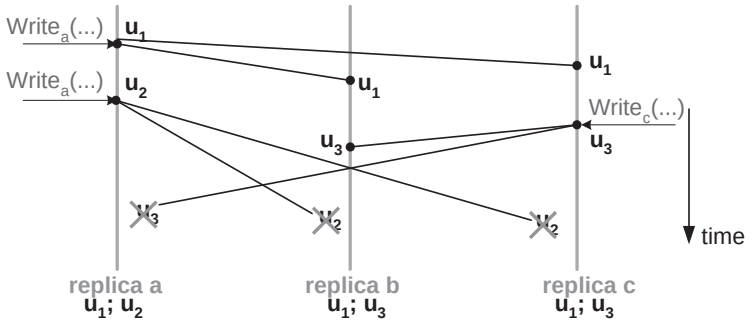


Figure 2.5: An example of strict happens-before ordering of updates.

before ordering ensures that no update, locally unknown at the time the client issued the write operation, will ever be interleaved between the new update and the previous ones. This property is very desirable, as it matches the common intuition of users of most applications, which find it in typical centralized and pessimistic systems. Nevertheless, it comes at the cost of discarding concurrent updates, in contrast to the previous weak consistency guarantees.

2.1.3 Eventual consistency

Eventual consistency (EC) [SS05] can be seen as a hybrid that combines weak and strong consistency guarantees. At its base, EC offers weak consistency guarantees, according to one of the previous criteria. However, EC also ensures that, eventually, the system will agree on and converge to a state that is strongly consistent. Typically, the criterion for strong consistence is sequential consistency.

As we describe in the following sections, most optimistic replication solutions offer EC. We provide a possible formal definition of EC in Chapter 3.

In EC, we may thus distinguish two stages in the life cycle of an update. Immediately after being issued, the update is optimistically ordered and applied upon a weakly consistent schedule of other updates. In such a stage, we say that the update resulting from the write operation is *tentative*, and the value that we obtain by executing the updates in the weakly consistent schedule is the current *tentative value* of the replica. Eventually, by means of some distributed coordination, the tentative update becomes ordered in some schedule that ensures strong consistency with the schedules at the remaining replicas, we call it *stable*; similarly, the value that results from the execution of such an ordering is the replica's current *stable value*.

Some systems are able to distinguish the portion of an ordering of operations whose corresponding updates are already stable, from the portion that is still tentative. These systems offer *explicit eventual consistency* (EEC). Systems with EEC are able to expose two views over their replica values: the stable view, offering strong consistency guarantees, and the tentative view, offering weak consistency

guarantees. Note that some systems offer EC but not EEC; i.e. they ensure that, eventually, replicas converge to a strongly consistent state, but cannot determine when they have reached that state.

EEC is a particularly interesting combination of weak and strong consistency guarantees. It can easily accommodate different applications with distinct correctness criteria and, consequently, distinct consistency requirements. Applications with stronger correctness criteria can access the stable view of replicated objects. Applications with less demanding correctness criteria can enjoy the higher availability of the tentative view.

2.1.3.1 Eventual Consistency with Bounded Divergence

While EC ensures strong consistence of the stable view, it allows the weakly consistent view to be arbitrarily divergent. For some applications, however, such an independence is not acceptable.

Eventual consistency with bounded divergence [YV00, SVF07] strengthens the weakly consistent guarantees of EC, according to some parameterized bounds. More precisely, it imposes a limit on the divergence that tentative values may have, using the stable values as the reference from which divergence is measured. Divergence bounds can be expressed in several ways; from the number of pending tentative updates, to semantic measures such as the sum of money from some account that is withdrawn by updates that are still tentative.

Operations that do not respect the parameterized bounds cannot be accepted and, hence, are either discarded or delayed. Hence, EC with bounded divergence offers lower availability than EC.

2.2 Limitations of Pessimistic Replication

Traditional replication employs pessimistic strategies, which prevent inconsistencies between replicas by reducing the availability of the replicated system [DGMS85, WPS+00].

Each site makes worst-case assumptions about the state of the remaining replicas. Therefore its operation follows the premise that, if any inconsistency can occur in result of some replica operation, that operation will not be performed. As a result, pessimistic strategies yield strong consistency guarantees such as linearizability or sequential consistency.

Before accepting an operation request at some replica, pessimistic replication runs some synchronous coordination protocol with the remaining system, in order to ensure that executing the operation will not violate the strong consistency guarantees. For instance, if the request is for a write operation, the system must ensure that no other replica is currently being read or written; possibly, this will mean waiting for other replicas to complete their current operations. In the case of a read operation, the system must guarantee that the local replica has a consis-

tent value; this may imply obtaining the most recent value from other replicas and, again, waiting for other replicas to complete ongoing operations.

In both situations, after issuing a local operation request, a client has to wait for coordination with other remote sites obtaining a response. Such a performance penalty is the first disadvantage of pessimistic replication. Further, if a network partition or a failure of some site should prohibit or halt the coordination protocol, then the request response will as well be disrupted. As a second shortcoming, pessimistic replication offers reduced availability if the latter occur with non-negligible frequency.

Thirdly, pessimistic replication inherently implies that two distinct write operations cannot be accepted in parallel by disjoint sets of sites. It is easy to show that, otherwise, we would no longer be able to ensure strong consistency guarantees. This poses an important obstacle to the scalability of pessimistic systems, as their sites cannot serve operations independently.

Some authors [SS05] point out a fourth limitation, that some human activities are not adequate to strong consistency guarantees. Rather, such activities are most appropriate to optimistic data sharing. According to such authors, cooperative engineering and code development are examples of such tasks, where users prefer to have continuous access shared data, even when other users are updating it, and possibly generating conflicts.

In the network environments that the present thesis considers, the availability and performance shortcomings clearly constitute strong reasons to not adopt pessimistic replication as our solution. Further, the fourth limitation is also very relevant for our objective of supporting collaborative activities, such as the ones we mention above.

While pessimistic replication is not the subject of our thesis, it is worth presenting three main approaches to achieve it, which we do in the following sections. As will become apparent later in the chapter, the protocols that optimistic replication uses to ensure eventual consistency inherit most ideas from these pessimistic replication approaches.

2.2.1 Primary copy

This approach [AD76, Sto79] assumes the existence of a single distinguished replica, designated as the primary copy. Every write operation to the logic object must be handled by the site holding its primary copy. Updates are then propagated to the remaining replicas.

In the case of reads, a lock has to be acquired at the primary copy's site. The actual read operations can then be performed at any replica of the logical object.

In the event of a network partition, only the partition containing the primary copy is able to access it. Upon recovery, updates are forwarded to the previously partitioned sites to regain consistency.

Under situations where network partition is distinguishable from the failure of a node, a new primary copy can be elected when the previous one fails. However,

if it is not possible to determine whether the primary copy is unavailable for failure or partition reasons, the system must always assume that a partition occurred and no election can be made.

2.2.2 Tokens

This approach [MW82] bears close resemblance to the primary copy scheme. The difference is that the primary copy of a logical object can change for reasons other than network partition. Each logical object has an associated token, which allows the replica holding it to access the object's replicated data. The token may circulate from site to site.

When a network partition takes place, only the partition that includes the token holder will thus be able to access the corresponding logical object. One disadvantage of this approach lies in the fact that the token can be lost as a result of a communication or site failure.

One variation of this basic token protocol is the *single writer multiple readers* [LH86], where two types of tokens are obtainable: read and write tokens. Several sites can simultaneously hold a read token on a particular logical object, which enables them to serve read requests to the local replicas of that object. On the other hand, a site can hold a write token, which implies that no other site holds a token of any type on the same logical object. With a write token, a site can serve both update and read requests to its local replica of the object.

2.2.3 Voting

In a voting strategy [Gif79], every replica is assigned some number of votes. Every operation upon a logical object must collect a read quorum of r votes to read from a replica or a write quorum of w votes to update a replica value.

The following conditions must be verified by the r and w quorum values:

(1) $r + w$ exceeds the total number of votes, v assigned to a logical object, and

(2) $w > v/2$

The first condition ensures that the intersection between read and write quorums is never empty. In a partitioned system, this means that a logical object cannot be read in one partition with read quorum and written in another partition with write quorum. This would lead to inconsistencies, since updated replicas in the write quorum partition would not be reflected in the values read in the read quorum partition.

The second condition guarantees that no more than one write quorum can be formed. This way, in the event of network partitions, the case of two or more partitions holding a write quorum for a logical object will never happen.

Notice that, if r is chosen so that $r < v/2$, it is possible for a logical object to be read by sites on more than one partition, in which case write operations are not allowed in any of such partitions. High availability for read operations can thus be achieved by choosing a small r value.

One drawback of the quorum scheme is that reading from a replica is a fairly expensive operation. A read quorum of copies must be contacted in this scheme, whereas access to a single replica suffices for all other schemes, provided that the read lock or token has already been acquired.

2.3 Introduction to Optimistic Replication

Optimistic replication, in contrast to pessimistic replication, enables access to a replica without *a priori* synchronization with the other replicas. Access requests can thus be served just as long as any single site's services are accessible. Write requests that each local replica accepts are then propagated in background to the remaining replicas, and an a posteriori coordination protocol eventually resolves occasional conflicts between divergent replicas.

Optimistic replication eliminates the main shortcomings of pessimistic replication. Availability and access performance are higher, as the replicated system no longer waits for coordination before accepting a request; instead, applications have their requests quickly accepted, even if connectivity to the remaining replicas is temporarily absent. Also, optimistic replication scales to large numbers of replicas, as less coordination is needed and it may run asynchronously in background. Finally, optimistic replication inherently supports asynchronous collaboration practices, where users work autonomously on shared contents and only occasionally synchronize.

Inevitably, optimistic replication cannot escape the trade-off between consistency and availability (see Section 2.1). Optimistic replication pays the cost of high availability with the limitation of offering weak consistency guarantees. Even if only rarely, optimistic replicas may easily diverge, pushing the system to a state that is no longer strongly consistent (e.g. according to the criteria of linearizability or sequential consistency) and, possibly, has different values that are semantically conflicting (we will define semantic scheduling and conflicts in Section 2.6.1.2). Hence, optimistic replication can only enforce weak consistency guarantees as replicas unilaterally accept tentative requests.

The optimistic replication systems in which we are interested attempt to offer eventual consistency. They run some distributed agreement protocol that is responsible to eventually resolve conflicts and make replicas converge, pushing the system back to a strongly consistent state. Eventual consistency is the central challenge of optimistic replication. It may take considerable time, especially when system connectivity is low; it may only be possible at the expense of aborting some tentative work; and it may require substantial storage and network consumptions.

Saito and Shapiro [SS05] identify the following basic stages in optimistic replication:

1. Operation submission. To access some replica, a local user or application submits an operation request. When studying eventual consistency, we are

mainly interested in requests that update the object. Update requests may be expressed in different forms, depending on the particular application interface (API) that the replicated system offers. In general, an update request includes, either explicit or implicitly, some precondition for detecting conflicts, as well as a prescription to update the object in case the precondition is verified [SS05].

Internally, the replicated system represents each request by updates.[2]. Each site may execute updates upon the value of its replicas, obtaining new versions of the corresponding object. Additionally, each site stores logs of updates along with each replica.

Whether update logging is needed or not depends on numerous design aspects of optimistic replication, which we will address in the next sections. First, update logging may enable more efficient, incremental replica synchronization. Second, update logging may be necessary for correctly ensuring eventual consistency. Third, update logging is useful for recovery from user mistakes or system corruption [SFH+99], backup [CN02, QD02, SGMV08], post-intrusion diagnosis [SGS+00], or auditing for compliance with electronic records legislation [PBAB07].

2. Update propagation. An update propagation protocol exchanges new updates (resulting from the above stage) across distributed replicas. The update propagation protocol is asynchronous with respect to the operation submission step that originated the new updates; potentially, it may occur long after the latter.

   Different communication strategies may be followed, from optimized structured topologies [RRP99] to random pair-wise interactions that occur as pairs of sites become in contact [DGH+87]. We call the latter approach epidemic update propagation. Its flexibility is particularly interesting in weakly connected networks, as it allows a site to opportunistically spread its new updates as other sites intermittently become accessible.

3. Scheduling. As a replica learns of new updates, either submitted locally or received from other replicas, the replica must tentatively insert the new updates into its local ordering of updates.

4. Conflict detection and resolution. Since replicas may concurrently accept new updates, conflicts can occur. In other words, it can happen that some replica, after receiving concurrent updates, cannot order them together with other local updates in a schedule that satisfies the preconditions of all updates.

   Therefore, complementarily to the scheduling step, each replica needs to check whether the resulting ordering satisfies the preconditions of its up-

[2] In practice, the system may represent updates in various forms, as we discuss in Chapter 3

dates. Only then is the replica sound according to the semantics of its users and applications.

If a conflict is found, the system must resolve it. One possible, yet undesirable, solution is to abort a sufficient subset of updates so that the conflict disappears. This solution has the obvious disadvantage of discarding tentative work. Some systems try to not to resort to such a solution, by either trying to reorder updates [KRSD01], by modifying the conflicting updates so as to make them compatible [TTP+95, SE98], or by asking for user intervention [KS91, C+93, CMP04].

5. Commitment. The update orderings at which replicas arrive are typically non-deterministic, as they depend on non-deterministic aspects such as the order in which updates arrived at each replica. Hence, even once all updates have propagated across the system, replicas can have divergent values.

   The commitment stage runs a distributed agreement protocol that tries to achieve consensus on a canonical update ordering, with which all replicas will eventually be consistent. We say that, once the value of a replica reaches such a condition, it is no longer tentative and becomes stable. Furthermore, we say that the updates that the replica has executed to produce such a stable value are *committed*.

   We should note that the moment where a replica value becomes stable is not necessarily observable. Recalling the definitions in Section 2.1.3, this distinguishes systems that offer eventual consistency (EC) from systems providing explicit eventual consistency (EEC). Systems where commitment is explicit are typically able to offer two views on a replicas, a tentative view, that results from the current update ordering, possibly including tentative updates; and a stable one, which is the most recent stable value. The former is accessible with high availability, while the latter is guaranteed to be strongly consistent across the stable views of the remaining replicas.

Optimistic replication is related to other topics in distributed systems, notably transactional database replication and group communication. Similarly to optimistic replication, some transactional database replication systems employ optimistic techniques for increased performance and availability [PL91]. As Saito and Shapiro note [SS05], transactional replication systems commonly target either single-node or well-connected distributed database systems, entailing frequent communication during transaction execution. This contrasts with the higher asynchrony and autonomy of optimistic replication [SS05].

Group communication [BvR94] may be employed to achieve data replication. Here, the common emphasis is on well-connected environments, where a group communication middleware ensures total ordered delivery of messages across the distributed nodes. Optimistic replication typically has weaker connectivity assumptions, and offers poorer guarantees regarding the order at which updates delivered to each replica.

For the above reasons, it is out of the scope of this chapter to survey transactional database replication and group communication.

We should also emphasize that it is not the goal of the present dissertation to address the security issues that may concern optimistic replication. Therefore, we assume a naive trust model which does not consider the existence of any distrusted entity. Solutions relying on more realistic trust models for replicated systems can be found, for instance, in [MSP+08], [RPCC93], [KBC+00], [RD01] or [BI03].

In the following sections, we analyze a number of fundamental design choices that affect the way optimistic replication may work and achieve eventual consistency.

2.4 Full versus Partial Replication

Data replication may be full or partial. Full replication requires that each site that replicates a given object maintain a replica of the whole value of the object. In contrast, partial replication, allows each replica to hold only a subset of the data items comprising the corresponding object. It may take advantage of access locality by replicating only the data items constituting the whole object that are most likely to be accessed from the local replica.

Partial replication is more appropriate than full replication when hosts have constrained memory resources and network bandwidth is scarce. Moreover, it improves scalability, since only a smaller subset of replicas needs to be involved when the system coordinates write operations upon individual data items (of the whole object). Nevertheless, achieving partial replication is a complex problem that has important implications to most phases of optimistic replication, from scheduling and conflict detection to update commitment. It is, to the best of our knowledge, far from being solved in optimistic replication.

Schiper et al. have formally studied the problem, and proposed algorithms that extend transactional database replication protocols[3] [SSP06], but not optimistic replication protocols. More recently, and also in the context of transactional database replication, Sutra and Shapiro have proposed a partial replication algorithm that avoids computing a total order over operations that are not mutually conflicting [SS08]. Hence, they claim to achieve lower latency and message cost.

Shapiro et al. analyze the problem of partial optimistic replication in [SBK04] in the context of the Actions-Constraints Framework formalism for optimistic replication, sketching directions for a potential algorithm. The PRACTI Replication toolkit [BDG+06] provides partial replication of both data and meta-data for generic optimistic replication systems with varying consistency requirements and network topologies. However, PRACTI does not support EEC, which strongly restricts the universe of applications for which PRACTI's consistency guarantees are effectively appropriate.

[3] In particular, the Database State Machine to partial replication approach [PGS03].

Full replication is a radically less challenging design choice. Not surprisingly, most existing solutions on optimistic replication rely on it.

2.5 Tracking Happens-Before Relations

Tracking happens-before relationships plays a central role in optimistic replication [SM94]. In different stages of optimistic replication we need to determine, given two updates (or the resulting versions), whether one happens-before the other one, or whether they are concurrent. For instance, when two sites replicating a common object get in contact, we are interested in determining which replica version happens-before the other one, or if, otherwise, both hold concurrent values. As a second example, systems that offer strict happens-before ordering of tentative updates need to determine whether two updates are related by happens-before or, instead, are concurrent.

A simple solution to both situations is to explicitly associate, to each update (or version) held by some replica, the identifiers of the updates (resp. versions) that happen-before it. This solution is adopted by some proposed systems [KWK03, KRSD01, PSM03, SBK04]. Of course, this solution is unsatisfactory as its space requirements grow linearly as each replica receives new updates. This implies storing, transferring and analyzing substantial amounts of data in order to be able to track happens-before relationships among updates/versions.

Tracking happens-before relations is a well studied problem, and a number of more space- and time-efficient alternatives have been proposed. All solutions aim at representing version sets (which in turn result from the execution of update sets) in some space-efficient manner that allows fast comparisons of version sets.

We address such solutions in the following sections.

2.5.1 Logical, Real-Time and Plausible Clocks

Logical clocks [Lam78] and real-time clocks consist of a scalar timestamp, used to express happens-before relationships. Each site maintains one such clock, and each update has an associated timestamp. If two updates are not concurrent, then the update with the lowest logical/real-time clock happens-before the other update. However, the converse is not true; i.e., neither solution can detect concurrency.

Logical clocks are maintained as follows. Every time the local site issues an update, it increments its logical clock and assigns such a timestamp to the new update. Every time a site receives an update from another site, the receiver site sets its logical clock to a value that is both higher than the site's current clock and the update's timestamp.

Real-time clocks assume synchronized (real-time) clocks at each site. The timestamp of an update is simply the time at which it was first issued. Real-time clocks have the advantage of capturing happens-before relations that occur outside system's control [SS05]. However, the need for synchronized clocks is a crucial

disadvantage, as it is impossible in asynchronous environments [CT96].

Plausible clocks [Val93, TRA99] consist of a class of larger, yet constant-size, timestamps that, similarly to logical and real-time clocks, can deterministically track happens-before relationships but not concurrency. Nevertheless, a plausible clock can detect concurrency with high probability.

2.5.2 Version Vectors

Version vectors, in contrast to the previous solutions, can both express happens-before and concurrency relationships between updates (and versions) [Mat89, Fid91].

A version vector associates a counter with each site that replicates the object being considered. In order to represent the set of updates that happen-before a given update, we associate a version vector to the update.

A usual implementation of a version vector is by means of an array of integer values, where each site replicating the object is univocally associated with one entry in the array, $0, 1, 2, ...$ Alternatively, a version vector can also be an associative map, associating some site identifier (such as its IP address) to an integer counter.

Each replica a maintains a version vector, VV_a, which reflects the updates that it has executed locally in order to produce its current value. Initially, all counters are set to zero in the replica's version vector.

When a new update is issued at replica a, the local site increments the entry in vv_{repA} corresponding to itself ($vv_{repA}[a] = vv_{repA}[a] + 1$). Accordingly, the new update is assigned the new version vector. As replica a receives a new update from some remote replica, b, the site sets $vv_a[i]$ to $maximum(vv_a[i], vv_b[i])$, where i is the site that issued the received update. Provided that updates propagate in FIFO order (see Section 2.1.2),[4], if $vv_a[b] = m$ (for any replica b), it means that replica a has received every update that replica b issued up to its m-th update.

Two version vectors can be compared to assert if there exists a happens-before relationship between the updates (or versions) they identify. Given two version vectors, vv_1 and vv_2, vv_1 *dominates* vv_2, if and only if the value of each entry in vv_2 is greater or equal than the corresponding entry in vv_1. This means that the update (resp. version) that vv_1 identifies happens-before the update (resp. version) with vv_2. If, otherwise, neither vv_1 dominates vv_2, nor vv_2 dominates vv_1, vv_1 and vv_2 identify concurrent updates (resp. versions).

Provided the number of sites is acceptably low, version vectors are very time-efficient. When generating and merging of replica versions, the new version vector is easily calculated by an increment or merge operation. A simple arithmetic comparison of the counters in two version vectors allows a system to conclude whether any of the associated replica versions dominates the other, or else, if a conflict exists.

An important limitation of version vectors is that the set of sites that replicate the object being considered is assumed static (hence, the set of replicas is

[4]This is also called the prefix property [PST+97].

static). Each site has an pre-assigned fixed position within a version vector. This means that replica creation or retirement is prohibited by this basic version vector approach. Secondly, version vectors are unbounded, as the counters in a version vector may grow indefinitely as updates occur. Finally, version vectors neither scale well to large numbers of replicating sites nor to large numbers of objects.

The following sections present extensions and alternatives to version vectors that eliminate most of the above limitations.

2.5.3 Version Vector Extensions For Dynamic Site Sets

Some variations of version vectors eliminate the latter's assumption of a static site set replicating an object.

A first solution was proposed in the context of the Bayou replicated system [PSTT96]. Bayou handles replica creation and retirement as special updates that propagate across the system by the update propagation protocol. Sites that create new replicas are directly added to the Bayou's version vector at each replica of the same object. Replica retirement, however, is a hard problem, which Bayou solves using an intricate recursive naming scheme to identify sites in each Bayou's version vector [PSTT96]. Such a naming scheme has the important drawback of an irreversible increase in replica identifier size as replicas join and retire.

A second extension of version vectors that supports dynamic replica sets is called dynamic version vectors [Rat98]. Dynamic version vectors are a version vector variant that is able to expand and compress in response to replica creation or deletion. Dynamic version vectors start as empty vectors and are able to expand with a new element as any replica generates its first update. The absence of an element for a given replica in a dynamic version vector is equivalent to holding a zero counter for that replica. Therefore, replica expansion is simple: when a given replica issues its first update, it simply adds a new entry to its affected dynamic version vector(s). The central challenge of dynamic version vectors is dealing with retiring replicas. In this case, when a replica retires, or when a replica (perhaps temporarily) ceases issuing updates, the system should remove the corresponding entry from all dynamic version vectors. However, the entry must be removed atomically among all sites in the system; otherwise, correctness would no longer be ensured [Rat98]. Ratner et al. solve the entry removal problem by running an epidemic agreement algorithm and assuming that permanent site failures are detectable. Agreement occurs once *every* correct replica acknowledges a removal proposal. Evidently, such an algorithm is extremely intolerant to temporary partitioning: any single inaccessible replica will halt the removal of an entry. Nevertheless, the entry removal algorithm does not stop the replicated system, which may transparently continue its regular operation, albeit with superfluous entries in its version vectors.

2.5.4 Version Stamps

Similarly to dynamic version vectors, version stamps [SCF02] express the happens-before relation in systems where replicas can join and retire at any time. Furthermore, version stamps avoid the need for a unique identifier per site. Assigning unique identifiers is a difficult when connectivity is poor[5], and sites come and go frequently.

Each replica constructs its version stamps, based on the updates and synchronization sessions that the local replica has seen. Version stamps consider three primitive operations: fork, update and join. A version stamp encodes the history of such operations that the local replica has seen. Using information that does not depend on any global information. This way, version stamps obviate the need for a mapping from each site to a unique identifier.

Maintaining and comparing version stamps is more complex than version vectors. Most importantly, the expressiveness of version stamps is limited. Whereas version vectors express happens-before relationships between any pair of versions/updates (including old versions that have already been overwritten), version stamps can only relate, the current versions of each replica. Almeida et al. designate such a set of coexisting versions as a *frontier* [SCF02].

This means that, in optimistic replication systems that maintain old updates in logs, one cannot use version stamps to determine whether a old, logged update happens-before some other update, as the former update may not belong to the same frontier as the latter. Version stamps are designed for systems that will only ever require comparing updates/versions that co-exist in some moment. This leaves out interesting log-based systems, which we will discuss in Chapter 3.

2.5.5 Bounded Version Vectors

Version vectors are unbounded data structures, as they assume that counters may increase indefinitely. Almeida et al. propose a representation of version vectors that places a bound on their space requirements [AAB04]. Like version stamps (see Section 2.5.4), bounded version vectors can only express happens-before relationships between versions in the same frontier.

They introduce an intricate mechanism, called bounded stamps, that replaces integer counters in a version vector. Such a space bound has the cost of additional complexity when manipulating and comparing version vectors. Moreover, similarly to version vectors, BVVs assume a static set of sites replicating the object corresponding to the version.

[5] Since referential integrity of site identifiers is hardly solved by distributed protocols or centralized name servers.

2.5.6 Vector Sets

Version Vectors represent the set of versions of a given individual replica. This means that the number of version vectors that a site needs to maintain and propagate grows linearly with the number of objects the site replicates. This is an obvious scalability obstacle, as most interesting systems (e.g. distributed file systems) have large numbers of replicated objects.

Malkhi and Terry have proposed Concise Version Vectors [MT05], later renamed Vector Sets (VSs) [MNP07], to solve the scalability problem of version vectors. VSs represent the set of versions of an arbitrarily large replica set in a single vector, with one counter per site, provided that update propagation sessions always complete without faults.

If communication faults disrupt update propagation sessions, VSs temporarily grow with the addition of complementary version vectors, called predecessor vectors. In the worst case, a VS will hold one predecessor vector per replica in the replica set that the VS represents; thus, tending to similar space requirements as version vectors. However, provided that communication disruptions are reasonably rare, VSs dramatically reduce the storage and communication overhead of version vectors in systems with numerous replicated objects [MT05].

Like version stamps and bounded version vectors, VSs can only express happens-before relationships among versions in the same frontier (see Section 2.5.4). This means that VSs, in their original form, are not applicable to systems need to maintain update logs along with their replicas, as is the case of many interesting systems that we present in this chapter. In particular, the replication protocol on which VSs are originally proposed [MNP07] does not ensure EEC.

2.6 Scheduling and Commitment

The road to eventual consistency has two main stages: first, individual replicas schedule new updates that they learn of in some way that is safe, i.e. free of conflicts; second, each such tentative schedule is submitted as a candidate for some a distributed commitment protocol, which will, from such an input, agree on a common schedule which will then be imposed to every replica.

In the following sections, we address each stage.

2.6.1 Scheduling and Conflict Handling

We distinguish two approaches for update scheduling: syntactic and semantic. The distinction lies on the information that is available to each replica when it is about to schedule a new update.

2.6.1.1 Syntactic Scheduling

In the syntactic approach, no explicit precondition is made available by the application that requests the operation causing the update. In this case, scheduling can only be driven by application-independent information such as the happens-before relation between the updates.

Based on such restricted information, syntactic scheduling tries to avoid schedules that may break users' expectations. More precisely, if update u_1 happens-before update u_2, then a syntactic schedule will order u_1 before u_2, as it knows that this is the same order in which the user saw the effects of each update. Note that, if the effects of both updates are commutative, the scheduler could also safely order u_2 before u_1, as the user would not notice it. However, as such a commutativity relationship is not known to the syntactic scheduler, it must assume the worst case where the updates are non-commutable. Hence, in the absence of richer information, syntactic scheduling is a conservative approach that restricts the set of acceptable schedules.

Concurrent updates, however, are not ordered. We find syntactic schedulers that behave differently in such a case. Some will artificially order concurrent updates, either in some total system-wide order (by real time or by site identifier, e.g. [TTP+95]) or by reception order, which may vary for each replica (e.g. [GRR+98, RRP99]). In either solution, the resulting schedule may no longer satisfy users' expectations, as the concurrent updates may be conflicting according to application semantics and, still, the scheduler decides to execute them.

Other syntactic schedulers opt for scheduling only one of the concurrent updates, and exclude the remaining concurrent updates from the schedule (e.g. [Kel99], [BF05a]). This approach conservatively avoids executing updates that can be mutually incompatible. However, it has the secondary effect of aborting user work, which is evidently undesirable.

Recalling Section 2.1, the former syntactic approach ensures happens-before ordering of updates, possibly combined with the total ordering, while the latter syntactic approach provides strict happens-before ordering.

2.6.1.2 Semantic Scheduling

In some cases, rich semantic knowledge is available about the updates. Essentially, such information determines the preconditions to schedule an update. The replicated system may take advantage of such preconditions to try out different schedules, possibly violating happens-before relationships, in order to determine which of them are sound according to the available semantic information.[6]

Some systems maintain pre-defined per-operation commutativity tables [JMR97, Kel99]. Whenever a replica receives concurrent updates, it may consult the table and check whether both correspond to operations that are commutable; if so, they can be both scheduled, in any order.

[6] We precisely define soundness in Chapter 3.

In other systems, operation requests carry explicit semantic information about the scheduling preconditions of each update. For instance, Bayou's applications provide dependency checks [TTP+95] along with each update. A dependency check is an application-specific conflict detection routine, that compares the current replica state to that expected by the update. The output tells whether the update may be safely appended to the end of such a schedule or not, which is equivalent to compatibility or conflict relations between the update and the schedule.

Even richer semantic information about scheduling preconditions may be available, as in the case of IceCube [KRSD01], or the Actions-Constraints Framework [SBK04]. Their approach reifies semantic relations between as constraints between updates. It represents such a knowledge as a graph where nodes correspond to updates, and edges consist of semantic constraints of different kinds.

We should mention that scheduling has important consequences in other stages of optimistic replication, notably update propagation and commitment. Consequently, although rich semantic knowledge permits higher scheduling flexibility, some systems partially abdicate it for efficiency of other stages of optimistic replication. For instance, Bayou imposes that scheduling respect the local issuing order, i.e. two updates issued by the same site must always be scheduled in that (partial) order. Such a restriction enables a simpler update propagation protocol, which can use version vectors to determine the set of updates to propagate [PST+97]. Inevitably, it limits scheduling freedom and, thus, increases the frequency aborts.

Semantic information may also help when no schedule is found that satisfies the preconditions of some update, by telling how the update can be modified so that its effects become safe even when the original precondition fails. In Bayou, updates also carry a merge procedure, a deterministic routine that should be executed instead of the update when the dependency check fails. The merge procedure may inspect the replica value upon which the update is about to be scheduled and react to it. Ultimately, the merge procedure can do nothing, which is equivalent to discarding the update.

Finally, other approaches are specialized to particular semantic domains and, using a relatively complex set of rules, are able to safely schedule updates, detecting and resolving any conflict that is already expected in such a semantic domain. Such approaches, however, are not directly generalizable to other domains.

A first example is the problem of optimistic directory replication in distributed file systems. The possible conflicts and the possible safe resolution actions are well studied, for instance in the algebra proposed by Ramsey and Csirmaz [RC01], or in the directory conflict resolution procedures of the Locus [WPE+83] and Coda [KS91] replicated file systems .

Other work follows the Operational Transformation method [SE98] to ensure consistency in collaborative editors. Instead of aborting updates to resolve conflicts, this method avoids conflicts by transforming the updates, taking advantage of well known semantic rules of the domain of collaborative editing. Operational Transformation solutions are complex. They typically assume only two concurrent users [SS05] and are restricted to very confined application domains. Research on

this method is traditionally disjoint with the work on general-purpose optimistic replication that this thesis addresses.

2.6.2 Update Commitment

Commitment is a key aspect of optimistic replication with eventual consistency. One may distinguish four main commitment approaches in optimistic replication literature.

2.6.2.1 Unconscious Commitment

A first approach may be designated as the unconscious approach [BGL+06]. In this case, the protocol ensures eventual consistency; however, applications may not determine whether replicated data results from tentative or committed updates. These systems are adequate for applications with weak consistency demands. For example, Usenet, DNS, Roam and Draw-Together [IN06] adopt this approach. The protocol proposed by Baldoni et al. [BGL+06] is another example.

2.6.2.2 Commitment Upon Full Propagation

Other approaches, however, allow explicit commitment. A second approach for commitment is to have a replica commit an update as soon as the replica knows that *every other replica* have received the update [Gol93]. This approach has two important drawbacks. First, the unavailability of any single replica stalls the entire commitment process. This is a particularly significant problem in loosely-coupled environments. Second, this approach is very simplified, in the sense that it does not assume update conflicts. Instead, it tries to commit all updates (as executed); no updates are aborted. The TSAE algorithm [Gol93] and the ESDS system [FGL+96] follow this approach, though by different means. One may also employ timestamp matrices [WB84, AAS97] for such a purpose.

2.6.2.3 Primary Commit

A third approach is a primary commit protocol [PST+97]. It centralizes commitment into a single distinguished primary replica that establishes a commitment total order over the updates it receives. Such an order is then propagated back to the remaining replicas. Primary commit is able to rapidly commit updates, since it suffices for an update to be received by the primary replica to become committed. However, should the primary replica become unavailable, commitment of updates generated by replicas other than the primary halts. This constitutes an important drawback in loosely-coupled networks.

Primary commit is very flexible in terms of the scheduling methods that may rely on primary commit. Examples of systems that use primary commit include Coda [KS91], CVS [C+93], Bayou [PST+97], IceCube [KRSD01] and TACT

[YV00]. In particular, to the best of our knowledge, existing optimistic replication solutions that rely on a rich semantic repertoire (namely, Bayou and IceCube) all use primary commit.

2.6.2.4 Voting

Finally, a fourth approach is by means of voting [PL88, JM90, AW96]. Here, divergent update schedules constitute candidates in an election, while replicas act as voters. Once an update schedule has collected votes from a quorum of voters that guarantee the election of the corresponding candidate, its updates may be committed in the corresponding order. Voting eliminates the single point of failure of primary commit.

In particular, Keleher introduced voting in the context of epidemic commitment protocols [Kel99]; his protocol is used in the Deno [CKBF03] system. The epidemic nature of the protocol allows updates to commit even when a quorum of replicas is not simultaneously connected. The protocol relies on *fixed per-object currencies*, where there is a fixed, constant weight that is distributed by all replicas of a particular object. Fixed currencies avoid the need of global membership knowledge at each replica, thus facilitating replica addition and removal, as well as currency redistribution among replicas.

Deno requires one entire election round to be completed in order to commit each single update [cKF01]. This is acceptable when applications are interested in knowing the commitment outcome of each tentatively issued update before issuing the next, causally related, one. However, collaborative users in loosely-coupled networks will often be interested in issuing sequences of multiple consecutive tentative updates before knowing about their commitment, as the latter may take a long time to arrive. In such situations, the commitment delay imposed by Deno's voting protocol becomes unacceptably higher than that of primary commit [BF05b].

Holliday et al. [HSAA03] have proposed a closely similar approach. Their algorithm also relies on epidemic quorum systems for commitment in replicate databases. However, they propose epidemic quorum systems that use coteries that exist in traditional (i.e. non epidemic) quorum systems, such as majority.

To the best of our knowledge, no study, either theoretical or empirical has ever compared Holliday et al.'s majority-based and Deno's plurality-based approaches. In fact, apart from the above mentioned works, epidemic quorum systems remain a relatively obscure field [BF08b].

2.6.3 Relation to Other Problems in Distributed Systems

The previous section described the main solutions for commitment that may be found in the context of optimistic replication. Out of such a scope, however, other work also relates to the problem of update commitment.

Lamport introduced the *distributed state-machine* approach for building replicated systems [Lam78], whereby all sites execute exactly the same schedule of

events. Considering updates as events, this is a solution for commitment. However, it does not consider conflicts.

The Paxos protocol [Lam98] computes a total order of events, relying on voting. Such a total order may be used to implement a distributed state-machine. Generalized Paxos [Lam05] extends the original Paxos protocol in order to take commutativity relations between events into account and compute a partial order. In such a case, the protocol is able to reach agreement with fewer messages.

In a comparable contribution, though in a different context, Pedone and Schiper propose two voting-based solutions for the Generic Broadcast problem [PS02]. They extend the problem of message ordering so as to consider commutativity between messages.

The problem of explicit commitment, as a means to achieve eventual consistency of an optimistically replicated system, is equivalent to the problem of consensus [FLP85]. Consensus solves commitment: if the tentative schedule of each replica is submitted as a proposal for consensus, the decided one may be committed. On the other hand, commitment solves consensus: if the set of consensus proposals is treated as a set of mutually conflicting updates, one for each proposal, and submitted for commitment, then the single update to commit (the others will abort) identifies the decided proposal. Consequently, the impossibility result [FLP85] of distributed consensus with one faulty process also applies to explicit commitment. Namely, this is present in every commitment solution mentioned previously.

2.7 Efficient Replica Storage and Synchronization

The issues of efficient replica storage and synchronization are fundamental for any realistic system using optimistic replication. In the next sections we discuss existing approaches to solve such problems in optimistic replication.

2.7.1 Operation Shipping

In some systems, applications provide a precise operation specification along each operation they request from the replicated system. We call such systems as operation-based or operation-transfer systems. Typically, operation-based requests are more space-efficient than the alternative of state-based requests, which instead contain the value resulting from the application of the requested operation. Systems such as Bayou [DPS+94] or IceCube [KRSD01], or operation transformation solutions [SE98] rely on operation-based requests.

2.7.2 Activity Shipping

Nevertheless, in many replicated systems, applications submit state-based requests, which may have considerable space requirements. Replicated file systems are a representative example, which support the traditional API of centralized file sys-

tems. In traditional file system APIs, applications request updates to file contents by providing new content values to be written upon the file.

Some proposed solutions try to extract operation-based requests from systems that are originally state-based [LLS99, CVS04]. This approach has been called activity shipping [CVS04]. Activity shipping tries to avoid network transference of the contents of a file that has been modified by, instead, propagating a representation of the operations that produced such modifications. With activity shipping, user operations are logged at a client computer that is modifying a file and, when necessary, propagated to the server computer. The latter then re-executes the operations upon the old file version and verifies whether the regenerated file is identical to the updated file at the client by means of fingerprint comparison.

Despite achieving higher reduction factors, activity shipping has the fundamental drawbacks of (a) requiring semantical knowledge over the accessed files and (b) imposing the operating environments at the communicating ends to be identical. In particular, the second requirement is often unrealistic in mobile and Internet environments, where device heterogeneity is a dominant characteristic.

2.7.3 Data Compression

Data compression is a vastly applied technique for dealing with memory or bandwidth limitations. The goal of data compression is to reduce the number of bits needed to store or transfer data, in order to increase effective data density [LH87]. Data compression algorithms employ some encoding scheme, which transforms some bit array into a smaller, compressed array.

Lossless data compression methods, such as the Lempel-Ziv algorithms [ZL78], may be used to reduce the storage demands of a file system. Examples include popular and prominent file systems such as NTFS [CW08] or ZFS [ZFS08]. Such file systems internally store file contents in a compressed state. They use efficient decompression and caching algorithms to ensure that such contents are transparently presented in their plain form to applications that access them via the file system API.

Similar lossless data compression algorithms are almost omnipresent in distributed protocols that need to efficiently transfer large amounts of data.

Compression algorithms can attain substantial space gains for most common workloads [LH87], traded-off for a performance penalty, as the compression and decompression algorithms need to be applied to the data to access from storage, or to transfer across the network.

Data compression algorithms are appropriate to work within relatively small content windows, such as individual files [LH87]. However, data compression algorithms do not scale to exploit redundancy within pieces of larger volumes of data such as entire disk partitions.

2.7.4 Data Deduplication

Data deduplication techniques try to detect chunks with identical contents within some data volume. In principle, data deduplication and data compression share the goal of exploiting data redundancy. However, they achieve such a goal through radically different approaches. Whereas data compression re-encodes data in order to increase its density [LH87], data deduplication eliminates duplicate chunks, either from storage or from network communication. Both can be transparently combined together.

Significant work has addressed the exploitation of duplicate contents for improved storage and network dissemination of state-based updates. We categorize proposed solutions into three main approaches, discussed in the following subsections.

2.7.4.1 Delta-Encoding

Systems such as CVS [C$^+$93], SVN [CMP04], Porcupine [SBL99], BitKeeper [HG02] and XDFS [Mac00] use *delta-encoding* (or *diff-encoding*) to represent each succeeding version of a replica. Starting from a base value of the replica, they encode succeeding versions by a compact set of value-based mutations (*deltas*) to the value of the preceding version. Replicas maintain some version tracking state (e.g. logical timestamps or version vectors). The exchange of such a state between a replica pair determines the minimal set of delta-encoded versions to be propagated between the pair. Delta-encoding can result in substantial storage and network gains [Mac00, AGJA06]. CVS and SVN are examples of popular systems that synchronize replicas by transferring only deltas instead of whole-value versions (in the case of CVS, such an optimization only occurs in the direction of the client, while it occurs in both directions in SVN).

Delta-encoding typically employs specific algorithms for detecting similarity, which are only applicable when the input is a pair of versions [HHH$^+$98]. Examples of *pair-based* algorithms include the copy/insert [Mac00] and insert/delete [MM85] algorithms, or the sliding-block algorithm [MM85]. The previous pair-based algorithms are not scalable to similarity detection among a large set of versions; this is not the case of similarity detection algorithms based on content-addressing, e.g. the compare-by-hash solutions we describe in Section 2.7.4.3.

Experimental studies [JDT05] suggest that, if one considers a given pair of versions, pair-based algorithms are, in some cases, able to detect more content similarity than multi-version compare-by-hash algorithms. This suggests that, in worst-case workloads where content similarity occurs *only* between succeeding versions of the same object, delta-encoding may actually behave better than compare-by-hash approaches (described next).

2.7.4.2 Cooperating Caches

Spring and Wetherall [SW00] propose a technique based on two cooperating caches at the ends of a network link connecting two sites. The technique relies on the fact that both caches hold the same ordered set of *n* fixed-size blocks of data, which correspond to the last blocks propagated between the sites. Before sending a new block, the sender site checks whether the local cache already holds the block. If so, the site knows that the block is also stored in the remote cache; thus, the site sends a token identifying the position of the block within the cache, instead of the block's contents.

The main limitation of cooperating caches is that cache size limits its ability to detect similarity between two sites. Additionally, a pair of cooperating caches can only exploit similarity among contents transferred between two sites. In larger systems, a site needs to maintain one cache per remote site.

2.7.4.3 Compare-by-Hash

A third approach is *compare-by-hash*. Compare-by-hash employs collision-resistant hash functions such as SHA-1 [Nat95] to detect redundant chunks of data. It relies on the low probability of a collision between the hash values calculated from the contents of two different chunks. It assumes that chunks with identical hash values have identical content.

Essentially, before transferring a set of values, the hash values of the chunks comprising the values are sent. Upon reception of such lightweight data, the receiver site checks if it already stores any chunks with hash values that are identical to any of the former hash values, by inspection of a local hash database. If so, it can immediately obtain the contents from local storage. The receiver site then replies with an identifier of the missing chunks, for which the hash comparison turned out negative; the contents of only such chunks are transferred across the network.

Systems such as the Low-bandwidth File System [MCM01], the Pastiche distributed back-up system [CN02], Haddock-FS [BF04], Shark [AFM05], Stanford's virtual computer migration system [TKS+03] and the OpenCM [SV02] configuration management system employ the above technique for efficient network transference. They divide values into contiguous (non-overlapping) chunks, either fixed-size or variable-size, using content-based chunk division algorithms as proposed by Muthitacharoen et al. [MCM01] or Eshghi and Tang [ET05].

Other systems use more intricate variations of the technique for higher efficiency. In the rsync utility [TM98], the recipient site (not the sender site) sends the hash values of the contiguous, non-overlapping fixed-sized chunks that form its current value of each object[7] to transfer. For each such object, the sender site compares the received hash values with the hash values of every *overlapping* fixed-size chunk of the current value that the site holds. This variation reduces the sensitivity of fixed-size chunk division to *insert and shift* operations. Experimental results

[7] In the case of rsync, objects are files.

[JDT05] suggest that such a variation detects more similarities than content-based chunk division for a sufficiently fine chunk granularity. However, such a variation applies only to similarity detection between a pair of values.

The TAPER replicated system [JDT05] optimizes the bandwidth of compare-by-hash with a four-phase protocol, each of which works on a progressively finer similarity granularity. Each phase works on the value portions that, after the preceding phase, remain labeled as non-redundant. A first phase detects large-granularity redundant chunks (whole file and whole directories), using a hash tree. A second phase runs the base compare-by-hash technique with content-based chunks. A third phase identifies pairs of very similar replicas (at each site) and employs rsync's technique upon each pair. Finally, a fourth phase applies delta-encoding to the remaining value portions.

The Jumbo Store [ELW+07] distributed utility service complements compare-by-hash by sending hash values organized into a generalization of a Merkle tree [Mer79], called HDAG. Nodes higher in the tree identify higher granularity chunks. By sending higher level hash values first, the receiver site may directly identify large-granularity chunks as redundant; it may therefore avoid the transference and comparison of the lower-level chunks comprising the higher level chunk. This approach shares resemblance with the first phase of TAPER.

Further analysis [Hen03, HH05] also argues that compare-by-hash is not as risk-free and advantageous as literature often assumes. Despite the low probability[8], hash collisions in compare-by-hash may occur, resulting in silent, deterministic, hard-to-fix data corruption [Hen03]. The delta-encoding and cooperating cache approaches are free from such a crucial limitation.

Compare-by-hash requires two network round-trips (one to compare hashes, another one to send contents). Furthermore, proposed optimizations of compare-by-hash (e.g. TAPER and Jumbo Store) add extra round-trips. In high-latency environments, the incurred delay may substantially reduce the benefit of the technique. In contrast, delta-encoding and cooperative caching always require only one round-trip, no matter how optimized the local similarity detection algorithm is and how many phases it involves.

2.7.4.4 Compare-by-Hash for Storage

The principle of compare-by-hash may also improve storage efficiency. Here, the technique consists in dividing values into chunks, whose contents are then stored in a repository and indexed by the corresponding hash value. The site then represents each value as an ordered list of hash values, which in turn reference individual chunks in the repository. Chunk redundancy is thus automatically exploited, since redundant chunks are only stored once in the repository. Examples of systems that adopt this technique include the Venti [QD02] archival system, Pastiche [CN02] and Haddock-FS [BF04].

[8] Different authors define contradicting expressions for such a probability. Namely, Muthitacharoen et al. [MCM01], Cox et al. [CN02] and Henson and Henderson [HH05].

Since this technique works locally, one may avoid false positives (due to hash collisions) at the cost of extra work: by complementing positive hash comparisons with a byte-by-byte comparison. The implementation of Venti employs this variant.[9] Pastiche and Haddock-FS, however, rely exclusively on hash comparisons and, hence, are prone to data corruption.

The chunk repository architecture, as we describe it above, is relatively inefficient for a number of aspects: (i) it requires extra indirect memory/disk accesses to read a value, (ii) it increases internal fragmentation, and (iii) by storing chunks in a randomly organized repository, it penalizes sequential read performance. Venti reduces such problems by storing chunks contiguously in a log-structured storage system [RO92] in order to maximize chunk contiguity. However, Venti's design does not support version removal from the log.

2.8 Complementary Adaptation Techniques

Complementarily to the core of optimistic replication protocols, some work focuses on the adaptation of optimistic replication to environments with constrained resources. Such an adaptation is key to the applicability of proposed systems to environments where, for instance, device memory or energy is limited, or network links are intermittent or have low bandwidth.

Fluid Replication [CN01] divides hierarchically-structured objects (file system trees) into smaller sub-objects, identifying the subtrees (represented by *Least Common Ancestors*, or LCAs) that are modified with respect to a base version. This technique exploits the temporal and spatial locality of updates in order to improve update propagation in networks that may be temporarily poorly connected. A client exchanges only meta-data updates (which include LCAs) with the server, deferring update propagation to moments of strong connectivity. The exchange of meta-data enables the client to commit updates it generates, even before their actual propagation to the server. Moreover, deferred propagation may be more network-efficient, as redundant or self-canceling updates in the batch of pending updates need not be propagated. As a drawback, Fluid Replication is limited to client-server systems.

Fluid Replication's separation of control from data is not new in optimistic replication. In Coda [KS91], servers send invalidations to clients holding replicas of an updated object. Update propagation to invalidated replicas is performed subsequently only. The Pangaea [SKKM02] file system floods small messages containing timestamps of updates (called harbingers) before propagating the latter. Harbingers eliminate redundant update deliveries and prevent conflicts. The Roma personal meta-data service [SKW+02] separates consistency meta-data management from data itself. However, such a management is for user presentation only, and is not integrated into the underlying replication protocol. Moreover, Roma is centralized and is designed for personal, not collaborative, scenarios.

[9] As mentioned in [HH05], even though an earlier paper [QD02] describes the system as exclusively reliant on hash comparisons.

Systems such as Bayou [PST+97], Rumor [GRR+98], or Footloose [PSYC03] allow an off-line form of anti-entropy. Off-line anti-entropy does not require the interacting replicas to be simultaneously accessible across the network. The sender constructs a packet containing enough information to emulate anti-entropy when the packet reaches the target replica. In networks with poor connectivity among replicas, off-line anti-entropy enables alternative means of anti-entropy. These are interesting when replicas are not mutually accessible, or if the network links connecting them are prohibitively expensive. For instance, alternative means include transportable storage media, as well as mobile and stationary devices, accessible through the network, that are willing to temporarily carry off-line anti-entropy packets.

Off-line anti-entropy has a higher communication overhead than regular anti-entropy. Whereas, in regular anti-entropy, the sender may ask the receiver what is the minimal set of relevant information to send, that is not possible in off-line anti-entropy. Therefore, packets exchanged in off-line anti-entropy should carry enough information so that they are meaningful for every potential receiver; hence, their size may grow significantly.

2.9 Case Study of Optimistic Replication Systems

Optimistic replication exists behind most Internet services and applications [SS05], which must cope with the limitations of a wide-area network. Examples include name and directory services, such as DNS [MD95], Grapevine [BLNS82], Clearinghouse [DGH+87], and Active Directory [Mic00]; caching, such as WWW caching [CDN+96, WC97, FGM+99]; or information exchange services such as Usenet [LOM94] or electronic mail [SBL99]. The semantics of these services already assume, by nature, weakly consistent data. In fact, the definition of their semantics was molded *a priori* by the limitations of wide-area networking; namely, slow and unreliable communication between sites.

Other optimistic replication solutions support more generic collaborative applications and semantics, the focus of the present document. Generic replication systems must cope with a wide semantic space, and be able to satisfy differing consistency requirements. In particular, they must accommodate applications whose semantics are already historically defined in the context of earlier centralized systems or pessimistically replicated systems in local area environments. Here, the optimistic replication system should provide semantics that are as close as possible to the original ones, corresponding to users' expectations.

In the next sections, we survey state-of-the-art general-purpose systems that rely on optimistic replication. At the end, we compare each solution's strengths and weaknesses.

2.9.1 Coda

Conventional distributed file systems, such as NFS [Now89] and AFS [MSC+86], can be regarded as replicated systems. Here, the replicated objects are files and directories that are replicated from the server's disks to client caches for better performance. These systems are based on a client-server architecture and employ pessimistic consistency policies.

Such design options are appropriate for operation over a fixed network, where the server infrastructure is always accessible to the wired clients. Network partitions and server failures are exceptional events that are expected to occur rarely. In this context, availability can be traded by stronger consistency policies with file system semantics that are closer or equivalent to those of local file systems. The popularity and wide use of these systems in fixed network domains is a symptom of their effectiveness.

However, when one allows the possibility of mobile clients, network partitions between such clients and the server infrastructure are no longer negligible [JHE99]. To achieve a better availability in the presence of such potentially disconnected clients, the Coda distributed file system [Sat02] enables clients to access the files stored in cache while being disconnected from the server machines [KS91].

2.9.1.1 Disconnected Operation

Coda inherits most of AFS's design options, including whole-file caching. Under normal operation, clients are connected to the server infrastructure and AFS's pessimistic cache consistency protocol is employed. When a client becomes disconnected from the servers, however, it adopts a distinct mode of operation. An optimistic consistency strategy is then used to enable the disconnected client to read and update the contents of the locally cached files and directories. A user can thus work on the documents cached on his disconnected mobile device while he is away from his wired desktop PC.

A client, or *Venus* in AFS terminology, can be in one of three distinct states throughout its execution: *hoarding*, *emulation* and *reintegration*. The client is normally in the hoarding state, when it is connected to the server infrastructure and relies on its replication services. Upon disconnection, it enters the emulation phase, during which it logs update operations to the cached objects. When a connection becomes available again, reintegration starts and the update log is synchronized with the objects stored in the servers' disks. The client then enters the hoarding state.

2.9.1.2 Hoarding

A crucial factor on the effective availability during disconnected operation is the set of objects that are cached. If, at the moment of disconnection, the set of cached files contains the files that the user will access in the future, disconnected operation can successfully achieve the desired availability. If, instead, the set of cached files

does not include those files that the mobile user will work on during the emulation phase, cache misses will occur, therefore disrupting normal system operation.

For the purpose of selecting the set of files that should be cached during the hoarding phase, in anticipation for a possible disconnection, Coda combines two distinct strategies. The first is based on implicit information gathered from recent file usage, by employing a traditional *least recently used* cache substitution scheme. Secondly, explicit information is used from a customized list of pathnames of files of interest to each mobile user, configured in a *hoard database*. Those files have an initial cache priority that is higher than the remaining cached files, in order to meet users expectations stated in the hoard database.

2.9.1.3 Conflict detection and resolution

Disconnected operation may lead to replica divergence. This raises the problems of conflict detection and resolution that characterize any optimistic replication approaches. The reintegration phase handles such problems.

Being a file system, Coda has to deal with two basic types of replicated objects: directories and files. In contrast with files, directories have well known semantics, which can be exploited by the file system to automatically maintain their integrity in the face of conflicting updates at different replicas. Coda solves the problem of directory inconsistency by a semantic approach that detects and automatically resolves conflicts. An example of such conflicts is when two disconnected clients each create distinct files on a common replicated directory. This will cause a conflict when the reintegration phase of both clients happens, since the directory replicas at each client have been concurrently updated. Using semantic knowledge, however, Coda easily solves this conflict by including both newly created files on the merged directory.

In the case of files, though, no semantic knowledge is available about their contents. For this reason a syntactic approach, based on version vectors, is adopted. Each replicated file is assigned a *Coda Version Vector* (CVV), which is a version vector with one element for each server that stores that file.

When a modified file is closed by a client, each available server holding such a file is sent an update message, containing the new contents of the file and the CVV currently held by that client, CVV_u. Each server i checks if CVV_u dominates CVV_i (see Section 2.5.2), where CVV_i is the CVV held by server i for the modified file. If so, the new update is applied at the server's replica and its associated CVV is modified to reflect the set of servers that successfully applied the update. If, otherwise, a conflict is detected at some server, that file is marked as *inoperable* and its owner is notified. If the client is operating in disconnected mode, this procedure is postponed until the reintegration phase.

2.9.1.4 Overall Considerations

Disconnected operation as defined by Coda still depends strongly on the server infrastructure, since the updates made during disconnection will only be available to other clients after reconciliation with the server. This may be acceptable in cases where file sharing between mobile clients are rare situations or disconnection periods are short and infrequent. However, in weak connectivity scenarios, Coda's disconnected operation model is inadequate, since the updates made in disconnected mode must be first propagated to the server before being visible to other clients.

In addition to disconnected operation at clients, Coda also enables optimistic read-write server replication in order to achieve better fault-tolerance in the face of server failures.

2.9.2 Concurrent Versions System and Subversion

The Concurrent Versions System (CVS) [C+93] is a version control system for collaborative work over a file system. Similarly to Coda, it lets clients locally replicate the files of a collaboration project. Clients work directly on file replicas using standard applications on a local file system. CVS does not offer the transparency of Coda, as it is not a replicated file system. Instead, users must interact with CVS by explicit commands; for example, it is the user responsibility to trigger update propagation to the server and respective commitment.

When a user propagates updates to the server replica (called a *repository*), the server tries to commit them. In case no concurrent work to the same file has already been committed, updates are committed. If, instead, concurrent work exists, the server checks whether the modified file portions overlap. If not, the client may optionally merge the concurrent work with the new updates, committing the latter. Otherwise, CVS notifies the user of a conflict; the user should then resolve it and try to commit again his or her work.

Subversion (SVN) [CMP04] is a successor to CVS, which fixes some design and implementation flaws of CVS. Among its main distinctive features, SVN supports data deduplication of transferred versions using delta encoding in both directions (see Section 2.7), versioning of file and directory meta-data updates, and eliminates the possibility of corruption when the commit protocol is interrupted.

2.9.2.1 Overall Considerations

Similarly to Coda, CVS and SVN centralize update propagation on the server infrastructure. Therefore, they are limited solutions for collaboration within groups of users disconnected from, or poorly connected to, the server infrastructure. Collaboration in poorly connected environments calls for higher flexibility, so that consistency may also progress through direct client-to-client interactions.

2.9.3 Gossip Framework

Some systems offer greater flexibility than the previous centralized example by relying on epidemic replication protocols [DGH+87]. One important example is proposed by Ladin et al. [LLSG92].

The Gossip framework provides highly available optimistic replication services to applications with weak consistency requirements. Furthermore, adaptability to applications with stronger consistency needs is provided by differentiated update types, with different availability levels.

The Gossip framework considers three possible update ordering modes: causal, forced and immediate. Causal mode orders updates after the updates that causally precede them. Causality relations are stated explicitly by the application that submits operation requests, which specifies the set of operations on which the new operation depends. Internally, version vectors represent the causal dependencies of each update.

Forced and immediate updates are guaranteed to be totally and causally ordered. The difference between the two is that forced updates are causally and totally ordered with respect to other forced updates, but only causally ordered within the remaining updates. In contrast, immediate updates are causally and totally ordered against all updates of all modes.

The choice of which ordering mode to use is left to the application. However, the two stronger ordering modes have significant costs on the effective availability. They require the site issuing an update to belong to a majority partition, which may significantly limit availability in poorly connected environments.

2.9.3.1 Update Propagation

Each replica maintains an update log and a version vector identifying the logged updates. Consistency between sites is achieved by the propagation of *gossip messages* between site pairs.

Gossip messages contain information about the updates known to a sender replica in order to bring the receiver replica up to date. Each message has two components: the version vector and a relevant segment of an update log of the replica of the sender site. The sender site determines the log segment to transfer by consulting a local *timestamp table* that stores, for each remote site, a version vector identifying the set of updates that site has already received. Such a table is maintained lazily; hence, its version vectors are conservative estimates. Therefore, a site may propagate updates that the receiver site already holds.

Updates are truncated from each site's log following a conservative scheme, based on the timestamps stored at each site's timestamp table. By comparing the update identifier with each version vector in the table, the local site can determine whether an update is guaranteed to have been received by all sites. Only in that case is the update pruned from the log.

2.9.3.2 Additional consistency guarantees

Clients of the replicated system can access any available site to issue read and write requests. Since relaxed consistency is employed between sites, a client can perform requests at sites with different replica values. Therefore it is possible that a client contacts a site whose replicas reflect less recent values than other sites which have already been read by the client. To prevent such inconsistent situations, the Gossip framework ensures that each client obtains a consistent service over time.

Each client maintains an associated version vector, which reflects the latest version the client accessed from a given replica. The client supplies such a version vector along with every request to any site. In the case of a read operation, the contacted site will only return the desired data value if the version vector of the request is dominated by the version vector of the current replica value. This condition guarantees that a client always retrieves data from a replica that is at least as new as the other replicas from which the client has already read from. This ensures that clients obtain a consistent service over time. Such a condition is equivalent to the *monotonic-reads session guarantee* which we address in Section 2.9.4.2, in the context of the Bayou replicated system.

In the case of write operations, a similar scheme is used. This is conceptually analogous to the *monotonic writes session guarantee* (Section 2.9.4.2), defined in the context of the Bayou replicated system.

2.9.3.3 Overall Considerations

The Gossip framework offers high availability replication services by providing weak (causal) ordering guarantees. When stronger consistency is required, applications may issue forced or immediate updates. Stronger consistency has costs in availability, since only sites in majority partitions may issue the forced and immediate updates.

The Gossip framework does not support basic features such as operations conflicts (it assumes that every pair of updates is compatible) or eventual convergence of the causal orderings to a strongly consistent state. Such absent features are fundamental conditions for many relevant applications.

The conservative log truncation scheme has the drawback that network partitions or failure of any individual site may cause updates to remain indefinitely in a site's log. Hence, log size can rapidly grow, as updates continue to be issued while older logged updates are not being discarded. For sites running at mobile devices, with typically poor memory resources, this is an important limitation.

2.9.4 Bayou

The Bayou System [DPS+94] is a mobile database replication system that provides high availability with weak consistency guarantees. Bayou employs a semantic approach that supports application-specific update conflict detection and resolution.

For that purpose, Bayou's programming interface requires applications to provide conflict detection and resolution instructions along with each update they request.

A Bayou client can issue updates at any accessible site. Hence, a highly available service is provided. Sites exchange received updates in pairwise interactions called *anti-entropy sessions*.

Bayou's semantic approach to conflict detection and resolution markedly differentiates it from weak consistency systems whose consistency protocol is based on a syntactic scheme (such as Coda, Rumor or Roam).

2.9.4.1 Conflict Detection and Resolution

Bayou uses a dynamic variant of version vectors (see Section 2.5.3) to syntactically identify and order replica versions. Tentative updates are epidemically propagated between sites and stored at each one's update log, ordered by the happens-before order defined by the version vectors of each update. Additionally, concurrent updates can be totally ordered according to the identifiers of the sites that issued each such update. Hence, Bayou ensures happens-before and, optionally, total ordering of tentative updates (see Section 2.1.2).

A semantic strategy complements the syntactic consistency scheme. Every update contains conflict detection and resolution instructions, respectively designated as *dependency checks* and *merge procedures*, specified by the application which issued the update. Such components of an update should reflect the issuing application's semantics.

A site executes an update's dependency check before applying it. A dependency check contains a query condition which is able to examine any part of the replicated database to determine whether a conflict exists or not. For instance, an update that books an appointment in a schedule database could check if, at the intended hour, any other event is already filling that time slot.

If the dependency check's query detects a conflict, the update's merge procedure is called. A merge procedure reacts to the conflict, producing a revised, non-conflicting update to be executed. Or, if no suitable alternative is found, the merge procedure can simply decide to abort the effect of the update.

One important aspect is that merge procedures are deterministic. This means that, if two sites schedule the same set of updates in the same order, their tentative replica values will be the same.

Being an optimistic replicated system, it is possible that a site receives, through anti-entropy with other peers, updates which are older than some of the tentative updates in the log. Therefore, such updates are placed in its correct position in local schedule, according to their version vectors. In such cases, conflict detection and resolution need to be performed, not only for each newly received updates, but also for the updates that follow them in the log.

2.9.4.2 Session Guarantees

To accommodate for applications with stronger consistency requirements, Bayou allows an application to place additional consistency guarantees, called *session guarantees* [DBTW94]. A *session* is an abstraction for a sequence of read and update operations performed on a database during the execution of an application. A session guarantee is selected by some application, Bayou ensures that the sequence of operations within the session's duration will meet the consistency requirements imposed by the session guarantee. Four session guarantees are offered: *Read Your Writes*, *Monotonic Reads*, *Writes Follow Reads* and *Monotonic Writes*. A detailed specification of each session guarantee can be found in [DBTW94].

To enforce session guarantees, Bayou can only accept an operation request at sites where reception of the request does not violate the currently selected session guarantees. For this reason, a session guarantee decreases availability.

2.9.4.3 Update Commitment

Bayou offers explicit eventual consistency (see Section 2.5.3), employing a primary commit scheme (see Section 2.6.2).

Bayou designates one site as the primary site. The primary site is responsible for the final ordering of the tentative updates it receives. Those updates then become stable updates. This ordering information is then propagated to the other sites by anti-entropy.

When a site receives learns that an update has become stable, it inserts that update at the tail of the stable prefix of the log. The corresponding tentative update is removed from the update log.

When a site inserts a new stable update in the stable prefix of the update log, all the tentative updates that follow it in the log must be undone. The new stable update is then applied, and its conflict detection and resolution procedures performed. Finally, all tentative updates that follow it are re-applied.

Update log truncation is dependent in the commit scheme. More precisely, a site is allowed to discard updates from its log as long as those updates have become stable [TTP+95].

2.9.4.4 Overall Considerations

Bayou's semantic approach to conflict detection and resolution enables it to combine concurrent updates in the same schedule, while ensuring that such a schedule is semantically sound. Consequently, in contrast to syntactic systems, Bayou can substantially minimize aborts in situations of update contention.

As a drawback, application programmers are now responsible for supplying dependency checks and merge procedures. Depending on the application's semantics, conflict detection code, along with their corresponding possible resolution actions, may severely increase the programming complexity. Furthermore, some

application semantics may have complex conflicts for which an appropriate resolution procedure may require an external decision from the user. Since merge procedures must be deterministic, user input is not allowed. The effectiveness of Bayou's semantic approach is, therefore, restricted to application domains with relatively simple underlying data semantics.

Secondly, Bayou relies on centralized scheme for update commitment. Hence, progress towards eventual consistency is dependent on a single point of failure. The system, being optimistic, remains highly available for accepting tentative updates. However, temporary inaccessibility or failure of the primary replica means that a user will wait longer before knowing if her tentative update was finally committed and will no longer rollback.

Finally, the time overhead of running dependency checks and merge procedures can be a relevant performance penalty. In particular, when a site receives an update and inserts it in the middle of the update log, the dependency checks, and possibly merge procedures, of the updates that follow it in the log must be re-applied.

2.9.5 OceanStore

Kubiatowicz et al. [KBC+00] propose OceanStore, a large-scale persistent storage infrastructure. OceanStore relies on an infrastructure of untrusted servers, which optimistically replicate data accessed by nearby clients.

The underlying replication protocol is based on Bayou's protocol, sharing most of its strengths and weaknesses. OceanStore enhances Bayou's protocol with redundancy and cryptographic techniques so as to protect stored data. In particular, a Byzantine agreement protocol [LSP95] replaces Bayou's primary commit protocol.

2.9.6 IceCube

IceCube [KRSD01] pushes Bayou's exploitation of semantics a step further. It attempts to minimize conflicts by seeking optimized update schedules, based on update semantics. In contrast to the previously described systems, IceCube considers more flexible orderings that may even violate happens-before. This follows from the observation that updates ordered by happens-before may still be semantically commuting, in which case the happens-before relation is an avoidable constraint on the space of possible orderings. IceCube is based on an explicit representation of semantics that is based on a graph of constraints between updates. Later work generalizes such a semantic model in the Actions-Constraints Framework [SBK04].

Similarly to Bayou, IceCube relies on a central site to compute an optimized stable schedule. Hence, its fault-tolerance is far from adequate to weakly connected systems.

2.9.7 Roam

Roam [RRP99] is an optimistically replicated file system intended for use in mobile environments. Roam allows any site to serve operation requests, without the need for a centralized server. This contrasts with Coda's model of disconnected operation (Section 2.9.1), in which all updates must first propagate to a server machine that then propagates them to other clients.

Roam is an extension to the Rumor file system [GRR+98], which in turn borrowed much of the replica consistency mechanisms from the Ficus file system [GHM+90]. Roam operates at application level, relying on the local file system services to store each replica.

Gossip messages propagate information about the updates that each replica has received so far to the remaining replicas. Replica consistency is achieved by performing periodic reconciliation sessions between peers in which gossip messages are exchanged.

2.9.7.1 Conflict detection and resolution

In Roam, directory conflicts are automatically resolved by taking into account their well know semantics. In the case of files, scheduling is purely syntactic, using dynamic version vectors (see Section 2.5.3). File replicas store no update log, just the current tentative value. Roam assigns each replica with a dynamic version vector and the consistency protocol ensures strict happens-before ordering guarantees (see Section 2.1.2.4).

Periodically, each site reconciliates its replicas with another site, according to the reconciliation topology imposed by the ward model, which we describe later. The reconciliation procedure itself is divided into three phases: *scan*, *remote-contact* and *recon*.

During the scan phase, the site examines its file replicas to check for new versions and, if necessary, updates their version vectors. This is done by a simple comparison of file modification times, which are obtained by querying the local file system. If the current modification time of a file replica is greater than that obtained during the last reconciliation, then a new version exists and the replica's version vector is updated to reflect it.

In the second phase, a remote site is contacted, according to the current ward topology, and asked to perform a similar version detection on its set of file replicas. As a result, the list of the file replicas and corresponding version vectors of the remote site is sent back to the requesting site.

Finally, version vectors of the files that both sites replicate in common are compared during the recon phase. If the remote version of a file dominates the local version, its entire content is transferred from the remote peer to update the local replica value. If, instead, the local version dominates the remote version, no action is taken since reconciliation in Roam is a one-way operation.

There is support for integration of type-specific file reconciliation tools into the

conflict resolution mechanisms. In the case of concurrent versions of files of types for which a reconciliation tool has been installed, a site automatically resolves the conflict. Otherwise, the user is notified of the conflict by email.

2.9.7.2 Ward model

Roam's architecture focuses on providing improved scalability by adopting a hierarchical reconciliation topology, called the *ward model* [RRPK01]. It groups nearby peers into domains called *wards* (wide area replication domains), according to some measure of proximity. Each ward has an assigned ward master. The result is a two-level hierarchical topology.

Reconciliation amongst wards is performed exclusively between ward masters. Inside each ward, all its ward members are configured into a conceptual ring. Such a ring topology imposes that each ward member reconciles only with the next ring member. The ring is adaptive, in the sense that it reconfigures itself in response to changes in the ward composition. Therefore, scalability is enhanced.

2.9.7.3 Overall Considerations

Roam provides a decentralized service, intended for mobile networks. By using an optimistic approach, any mobile site is able to accept operation requests, which enhances availability.

Replica consistency relies on an epidemic propagation of updates between sites. Roam adopts a two-level hierarchical topology, in which sites are grouped into wards. Update propagation is achieved by reconciliation sessions between pairs of sites within each ward and between ward masters. The result is increased scalability. Such an architectural model, which dynamically adapts itself to reflect the proximity amongst sites, is particularly effective in supporting scenarios where ad-hoc co-present groups of mobile devices are frequently formed.

The consistency protocol itself is relatively simple. The absence of an update log avoids the need to keep track of each update issued to the replicated file system objects. Instead, new replica versions are detected only during periodic reconciliation phases, by analysis of their modification times. Moreover, since no update log is maintained, Roam has a low memory overhead. Such a factor is especially important for mobile devices, which typically have poor memory resources.

Roam's weak consistency guarantees are a crucial drawback. Roam's consistency protocol does not regard any notion of update commitment. Therefore, data is always tentative. This limitation restricts Roam's applicability to applications with sufficiently relaxed correctness criteria.

2.9.8 Deno

The Deno system [CKBF03] supports optimistic object replication in a transactional framework for weakly connected environments. Deno is a pessimistic system. Nevertheless, Deno could be easily extended to an optimistic system.

In Deno, updates represent transactions, which commit if the update commits, and abort in case a concurrent update commits. Deno achieves replica consistency through an epidemic update propagation scheme. At any time, conflicting tentative updates can be issued at distinct replicas. Deno's consistency protocol is responsible for reaching a consensus amongst all sites on which one of such conflicting, tentative updates is to be committed. A distinguishing feature of Deno's design is that it relies on an epidemic voting approach (Section 2.6.2) to achieve such a goal.

2.9.8.1 Elections

Deno regards update commitment as a series of elections. Each election decides, amongst a collection of concurrent tentative updates, which one of them should be accepted as stable while the remaining updates are aborted. Each site acts as a voter in such elections. Similarly, each tentative update acts as a candidate for one election. Once an election is over, a new election is started.

Each site, or voter, has an assigned *currency*, which determines that site's weight during each voting round. An invariant of the system is that there is a fixed amount of currency, 1.0, which is divided among all sites of a given object. Currencies are not necessarily distributed uniformly among sites. Neither is their distribution static, since sites can perform currency exchange operations [CK02].

Voting takes place in a decentralized manner, relying on epidemic information propagation between voters to reach an eventual election result. Each site thus determines the result of each election, in function of the local information it has received from its peers. A site unilaterally decides that an election has terminated when it knows that some candidate has collected enough votes that guarantee a plurality, no matter how the votes that the site still does not know about get distributed.

Epidemic propagation of vote and election information ensures that all sites will eventually decide the same outcome for every election. Sites propagate to other sites the voting information they are aware of, which includes: (i) the elections that have already terminated and their outcomes, and (ii) the votes that have been cast in the current election. The receiving voter then updates his election state according to such information. Additionally, if the receiving voter hasn't yet cast a vote, he automatically votes on the same candidate which has been voted by the voter which initiated the anti-entropy session.

2.9.8.2 Conflict Handling

Deno exploits application semantics by means of optional commutativity information about the operations it receives. Such information may exist in the form of a pre-defined *commutativity table*, which relates abstract classes of operations that are commutable; or in the form of *commutativity procedures*, which are provided by applications along with each submitted transaction and are similar to Bayou's dependency checks.

Each site consults the commutativity data structures whenever one or more updates lose an election. Before aborting such updates, the site checks whether such updates are compatible with the update that has just won the election and, hence, committed. If so, the loosing, yet compatible, updates are re-submitted as candidates for the subsequent election, instead of aborting.

Therefore, provided no conflicting updates updates occur, all concurrent and compatible updates may eventually commit, one at each election.

2.9.8.3 Overall Considerations

Deno provides highly available replication services, designed for use in mobile and weakly connected environments. The main contribution of Deno's protocol design is the adoption of an epidemic voting scheme for update commitment. In contrast to a primary commit scheme, a voting scheme does not rely on a single site. Instead, a quorum of sites is required to decide which update, amongst a collection of concurrent updates, is to be committed.

The drawback of the voting scheme is its overhead. In fact, when connectivity is good, defining one site as a primary server improves update commitment and network communication w.r.t. using Deno's voting scheme [Kel99].

Furthermore, in situations where sites construct sequences of multiple tentative updates, each one happening-before the next one, Deno requires one election per update before committing the whole sequence. In contrast, the primary commit scheme can atomically commit the whole sequence with the same number of messages as it would require for a single update.

2.9.9 AdHocFS

AdhocFS [BI03] is a distributed file system designed for supporting pervasive computing in mobile ad-hoc environments [CM99]. The system is based on the premise that, in such scenarios, a fixed server infrastructure that provides the file system's services may not always be available. Therefore, AdHocFS's goal is to effectively support information replication between mobile users in the absence of such an infrastructure.

Furthermore, AdHocFS distinguishes between situations where mobile devices are working in isolation from any other devices and situations where groups of mutually accessible mobile devices cooperatively share and manipulate information. To deal with the distinct characteristics of each such scenario, AdHocFS uses distinct replica consistency strategies for each case, as we describe next.

2.9.9.1 Home Servers and Ad-Hoc Groups

AdHocFS's architecture considers the existence of a trusted stationary server infrastructure and a collection of wireless mobile devices that may be frequently disconnected from such an infrastructure.

Each file has a replica stored on a site located on the server infrastructure, which is designated as the *home server* for that file. The replica on the server infrastructure acts as a primary replica, in a primary commit protocol. Mobile devices can obtain a replica of the files stored at a home server when a connection is available. Upon disconnection, mobile devices tentatively operate upon the local replicas using a optimistic replication strategy.

When disconnected from the fixed infrastructure, mobile devices constitute *ad-hoc groups*. An ad-hoc group is a collection of mobile devices which are mutually accessible within one-hop wireless links. One extreme case is a singleton group, in which a group is formed by only one isolated device. AdHocFS dynamically manages the membership changes of ad-hoc groups as they are merged or separated.

This ad-hoc group model shares some similarities with the ward model of the Roam replicated system (Section 2.9.7). However, whereas the latter groups for better scalability, the AdHocFS also explores the high connectivity that exists among the sites within an ad-hoc group for consistency purposes.

In fact, AdHocFS employs a token-based pessimistic protocol (Section 2.2.2) within the members of each ad-hoc group, complementing the global optimistic consistency protocol. As a result, sequential consistency is accomplished if one only regards the set of replicas located in each ad-hoc group.

2.9.9.2 Overall Considerations

Using the token strategy of AdHocFS, updates are held back until a write token is granted to writer site. That, in turn, won't happen until all the preceding updates have been received by that site. Therefore, update concurrency within an ad-hoc group is eliminated. Of course, inter-group update concurrency, and potentially conflicts, may still occur. However, the overall global conflict rate is considerably reduced in situations where substantial update activity happens in the context of co-present ad-hoc groups.

AdHocFS has three main drawbacks. First, only the home server provides access to the stable value of some file. The mobile sites hold only tentative replica values, where tentative updates are immediately applied upon being received. Hence, mobile users that are disconnected from the home server are only able to access the tentative version of files.

Additionally, the home server acts as the primary replica of a primary commit scheme (Section 2.6.2). As discussed in the previously in the chapter, network partitions or failure of the home server may disrupt update commitment.

Finally, a AdHocFS uses an explicit version history (analogous to the first solution in Section 2.5) to represent the tentative updates at each replica. This approach requires that, in order to determine the prefix relationship between two replicas, the contents of the version histories of both replicas be thoroughly examined. This typically compels one version history to be entirely transferred to the other site in order to determine if the prefix relationship is met. Such a requirement is particularly expensive in mobile environments, where network bandwidth is normally

scarce.

2.9.10 TACT and VFC

Yu and Vahdat propose TACT [YV00], a middleware layer for support of wide-area collaborative applications. TACT assumes that each update has an associated weight, which may be defined by applications that issue write requests. TACT complements weak consistency guarantees with bounds on consistency according to three metrics, *Numerical Error*, *Order Error*, and *Staleness*. Numerical Error limits the weight of updates that can be applied across all replicas before being propagated to a given replica; Order Error limits the weight of tentative updates that can be waiting for commitment at any one replica; and Staleness places a real-time bound on the delay of update propagation among replicas. As a result, TACT captures the consistency spectrum lying between the strong consistency guarantees offered by pessimistic replication and the weak consistency guarantees of unbounded eventual consistency.

Vector Field Consistency (VFC) combines and extends sophisticated consistency models as TACT's with the notions of locality-awareness [SVF07]. In VFC's model, objects are positioned within an N-dimensional space. At each moment, the guaranteed consistency of an object replica depends on the current distance of the object to a *pivot*. Distributed games for ad-hoc networks are one example of an application of VFC.

To the best of our knowledge, all proposed instantiations of either model [YV00, SVF07] rely on a centralized site for commitment. This dependence restricts the applicability of the bounded divergence model in weakly connected environments.

2.10 Summary

This chapter surveys optimistic replication, a basic tool for supporting collaborative activity in weakly connected environments. We identify fundamental design choices in optimistic replication, and describe state-of-the-art design alternatives. Namely, we address consistency guarantees and their trade-off against availability; the problem of partial replication; mechanisms for tracking the happens-before relation among updates and versions; approaches for scheduling and commitment; techniques for efficient replica storage and synchronization; and complementary adaptation mechanisms.

We claim that, although much research has focused on optimistic replication, existing solutions fail to acceptably fulfill the crucial requirements for most applications and users in weakly connected and resource-constrained environments. Namely, such requirements are rapid update commitment, fewer aborts and adaptation to network and memory constraints.

We support such a claim by studying relevant state-of-the-art systems that rely on optimistic replication, mapping them against the previous design alternatives.

System	Replicated Objects	Replic. Type	Replication strategy	Version tracking	Topology
Coda	Files and Directories	Full	Optimistic in disconnected operation. Pessimistic when connected (assuming no optimistic server replication).	Version vectors	Client-server
CVS	Files	Full	Optimistic	Real-time/logical stamps	Client-server
SVN	Files and directories	Full	Optimistic	Real-time/logical stamps	Client-server
Bayou	Database	Full	Optimistic	Bayou's version vectors	Any
OceanStore	Database	Full	Optimistic	Bayou's version vectors	2-tiered peer-to-peer
IceCube	Database	Full	Optimistic	Explicit constraint graph	Any
Roam	Files and Directories	Full	Optimistic	Dynamic version vectors	Ward model
Gossip framework	Database	Full	Optimistic	Version vectors	Any
Deno	Database	Full	(Optimistic)	Election counter	Any
AdHocFS	Files and directories	Full	Optimistic	Home server timestamp + update log contents	Any
TACT	Database	Full	Optimistic	Version vectors	Any
VFC	Objects	Full	Optimistic	Array of dirty blocks	Client-server

Figure 2.6: General characterization of analyzed systems.

Tables 2.6 to 2.8 compare the overall characteristics of each analyzed system.

As discussed earlier in the chapter and summarized in Tables 2.6 to 2.8, proposed optimistic replication solutions either do not support update commitment, or impose long update commitment delays in the presence of node failures, poor connectivity, or network partitions. Some commitment approaches are oblivious to any application semantics that may be available; hence, they adopt a conservative update commitment approach that aborts more updates than necessary. Alternatively, semantic-aware update commitment is a complex problem, for which semantically-rich fault-tolerant solutions have not yet been proposed. Finally, more intricate commitment protocols that aim at fulfilling the above requirements have significant network and memory overheads, which are unacceptable for environments where such resources are scarce.

System	Scheduling and conflict handling	Consistency guarantees	Commit. scheme
Coda	Semantic for directories. Syntactic for files, by default. Supports type-specific file mergers for semantic conflict resolution.	Disconnected: strict h-b ordering; connected: seq. consistency.	Primary commit.
CVS	Syntactic.	Tentative: strict h-b ordering; stable: seq. consistency.	Primary commit.
SVN	Semantic for directories. Syntactic for files.	Tentative: strict h-b ordering; stable: seq. consistency.	Primary commit.
Bayou	Semantic, using dependency checks and merge procedures per update; on top of syntactic scheduler.	Total + h-b ordering in tentative view. Seq. consistency in stable view.	Primary commit scheme.
OceanStore	Semantic, using dependency checks and merge procedures per update; on top of syntactic scheduler.	Total + h-b ordering in tentative view. Seq. consistency in stable view.	Byzantine agreement protocol.
IceCube	Purely semantic.	Eventual seq. consistency.	Primary commit scheme.
Roam	Semantic for directories. Syntactic for files, by default. Supports type-specific file mergers for semantic conflict resolution.	Strict h-b ordering.	No commitment
Gossip framework	Causal updates identify updates that causally precede them. No support for conflicting updates.	Causal ordering of causal updates. Total + causal ordering of forced and immediate updates. Roaming clients see monotonic reads and writes.	No commitment for causal updates. Majority multicast for forced and immediate updates.
Deno	Syntactic, complemented with semantic commutativity tables/procedures.	Strict happens-before ordering in tentative view. Seq. consistency in stable view.	Epidemic weighted voting.
AdHocFS	Automatic conflict resolution for directories. Syntactic approach for files.	Strict causal consistency in tentative view. Seq. consistency at home servers.	Primary commit.
TACT	Syntactic, complemented with dependency checks and consistency bounds.	Bounded divergence, eventual seq. consistency.	Primary commit.
VFC	Syntactic with multi-dimensional bounded divergency	Bounded divergence, eventual seq. consistency.	Primary commit.

Figure 2.7: Overall comparison of consistency aspects of analyzed systems. (h-b and seq. consistency stand for happens-before and sequential consistency, respectively.)

System	Storage requirements per replica	Synchronization requirements
Coda	State-based Update log and stable value (cached files).	Incremental propagation of state-based updates.
CVS	Single value at client. State-based delta-encoded log at server.	Transfers compressed whole value to server; compressed delta to client.
SVN	Single value + log of deltas.	Compressed delta transference in both directions.
Bayou	Operation-based update log and stable value.	Incremental propagation of operation-based updates.
OceanStore	Operation-based update log and stable value.	Incremental propagation of operation-based updates.
IceCube	Operation-based log.	Incremental propagation of operation-based updates.
Roam	Single value.	Whole value transference.
Gossip framework	Operation-based update log and stable value.	Incremental propagation of operation-based logs.
Deno	Single value + operation-based candidate transactions.	Incremental propagation of operation-based candidate transactions.
AdHocFS	Single-value.	Propagates modified blocks only.
TACT	Operation-based.	Incremental propagation of operation-based updates.
VFC	Single-value.	Whole-value transference.

Figure 2.8: Overall comparison replica storage and synchronization requirements of analyzed systems.

Chapter 3

Architectural Baseline

We start by describing the system model and the base architecture that underlie the remainder of the present thesis. The system model we assume is common to most related work on optimistic replication. Further, the base architecture borrows most design elements and decisions from the state-of-the-art (Chapter 2). It constitutes an initial solution, which lays the ground for the next chapters, which then introduce the contributions of the thesis. Each contribution extends or modifies parts of this initial solution, eliminating or minimizing some of its limitations. The generality of the underlying system model and architectural base makes our contributions widely applicable.

The remainder of the chapter is organized as follows. Section 3.1 presents main notational conventions that we follow throughout the thesis. Section 3.2 starts by defining the system model, stating the problem and proposing such an initial, yet limited, solution. Section 3.3 then gives an overview of the contributions of this thesis, illustrating how they overcome shortcomings of the initial solution.

3.1 Notation

In the remainder of the thesis, we use the following notation:

- When referring to sites, we use symbols A, B, C, etc.
- When referring to replicas, we use symbols a, b, c, etc.
- When referring to objects, we use symbols x, y, z, etc.
- When referring to updates, we use symbols u, u_1, u_2, u_3, etc.
- When referring to versions, we use symbols v, v_1, v_2, v_C, etc.

The meaning of each notion above will be introduced in the remainder of this chapter.

Moreover, except where otherwise mentioned, all algorithms presented hereafter are assumed to run atomically. We make such an assumption for presentation simplicity, and without lack of generality.

3.2 Constructing an Baseline Solution

The following chapters propose an optimistic replication middleware service for support of distributed, collaborative applications. The replication service builds on a distributed system, composed of a number of computational nodes, which we designate *sites*. Sites may either be stationary or mobile nodes, and the system may not always be fully interconnected.

The system is asynchronous, and sites have no access to a global state or global clock. Sites may fail permanently, and only by crash failures.[1] Transient network partitions may also occur, restricting communication to processes inside the same partition.

3.2.1 Users, Applications and Objects

Each user performs her collaborative work using local applications running at some individual site. Applications, in turn, interact with the local replication service. From the user's and application's viewpoint, collaboration is extremely easy. It involves simple tasks such as creating objects, sharing them across the interested group of users, and accessing and modifying them as if they were local objects. Each such operation may be performed at any time and anywhere by any user with access to some individual site, independently of the remaining sites. The replication service hides any effort associated with supporting such operations.

An application may read from one of two views of the object: a *tentative* and a *committed* view. Write accesses are always performed on the tentative view. In general, accesses to the tentative view are weakly consistent: writes may later abort, and reads may reflect writes that will later abort. In contrast, data read from the committed view are guaranteed to be consistent across the system.

We impose very few assumptions on the objects and the applications. We consider generic objects. In practice, objects may be files and directories, database items, or Java objects, for example. Applications are external entities to the replication service, which make requests to the local replication service through some application programming interface (API). They may request to create objects, obtain local access to existing objects, and read from and write to the value of some object. Read and write operations are issued upon individual objects.

[1] Consequently, we assume byzantine failures do not occur.

3.2.1.1 Write Requests

Write requests are key to the operation of the replication service. Successive write requests cause a replica to have different values throughout time, each of which we call a *version*.

Concerning write requests, we make no assumption on (i) the way write operations are specified via the API, nor on (ii) what semantic information about the write is made available to the replication service.

Regarding the first aspect, we allow both *state-based* or *operation-based* representations. The state-based form is used for transparency, when no operation specification of the operation causing the write is available. A state-based write request includes the following components:

1. *Value*, an opaque array of bits;

2. *Offset*, an integer value that defines where, within the object value, the value is to be applied;

3. *Truncate*, a boolean value that indicates whether the previous value of the object is to be deleted prior to the write.

If, otherwise, a precise operation specification is passed along the write request, the request is in an operation-based form. For instance, in the case of database items, the application may represent a write request using an SQL data manipulation expression [CB74] (whereas a state-based form would specify the modified value of the affected database items).

Concerning the second aspect, write semantics, the API may provide information that allows the replication service to infer relations such as commutativity and conflict between the requested write and other writes. Among others, such information may allow the replication service to find out whether, according to application semantics:

- when executed after a given sequence of writes, the requested write produces a value that is semantically correct (compatibility) or not (conflict);

- if different orderings of compatible write requests produce values that are semantically equivalent (commutativity).

The above semantic information may be expressed by different means; for example, using Bayou's dependency checks [TTP+95] or the Actions-Constraints Framework [SBK04]. As will become clear shortly (in Section 3.2.5), such semantic information is useful for combining concurrent work from the distributed, collaborating users.

Evidently, the replication service also supports applications that do not provide any explicit semantic information along with their write requests. For instance, this

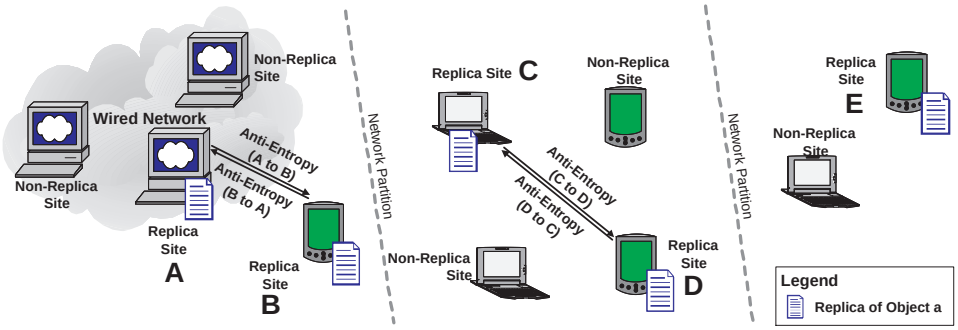


Figure 3.1: Illustration of system architecture. Mobile and stationary sites replicate object x and interact by pair-wise unidirectional anti-entropy.

is the case of already existing applications that use the replication service to access files via a standard file system interface, such as POSIX [POS90].

A final assumption is that local execution of write requests is recoverable, atomic and deterministic. The former means that the value of the object will not reach an inconsistent value if the site fails before the write completes. It follows from the other two properties that the execution of the same ordered sequence of writes at two copies of an object in the same initial value will yield an identical final value.

3.2.2 Base Service Architecture

We now focus on the base architecture of the replication service. In practice, each site maintains replicas of the objects that applications request. We assume that, for each object, the set of replicas does not change over time. Each site replicating the object has a unique identifier, and knows the identifiers of the remaining (static) set of sites replicating that object.

We assume full object replication. A site may dynamically choose which subset of objects it currently replicates.[2] However, for each object, the corresponding replica stores the complete value of the object.

Considering a given object, we may divide the universe of sites into those that currently replicate it and those that do not. We refer to the former as *replica sites* and to the latter as *non-replica sites*. Hereafter, when the discussion context clearly considers a particular object, we will sometimes use the term replica to interchangeably mean replica and the site holding the replica. We shall also use the term non-replica to designate a non-replica site of the replica being considered.

Sites communicate in order to synchronize replicas as the latter evolve due to application activity. For communication flexibility, we adopt the peer-to-peer architectural model among sites.

[2] In other words, the system does not support partial replication [JI01, HAA02, BDG+06, SSP06].

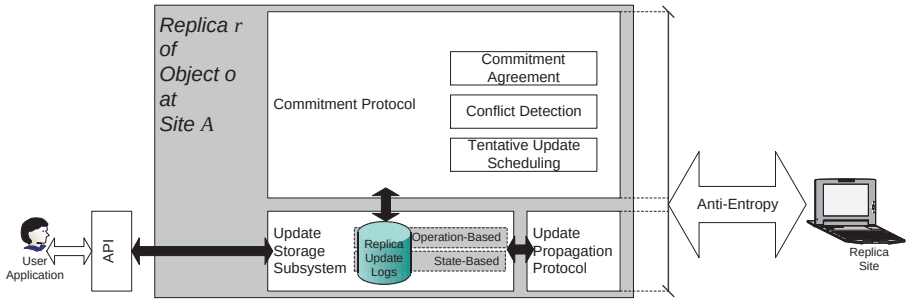


Figure 3.2: Functional components of the replication protocol at each site and their interactions with applications and other replica sites.

Consider a given object. When a replica site (of the object) gets in contact with another replica site, the former may initiate a pair-wise unidirectional synchronization session, called an *anti-entropy session* [DGH+87]. Figure 3.1 illustrates synchronization within a system of sites in a partitioned network, distinguishing replica sites from non-replica sites (considering a particular object, x). In an anti-entropy session, a replica site, A, sends to a replica site, B, any new data and/or meta-data concerning x that the replica at B stores does not hold yet.

The replication service is based on two functional components, (i) replica storage and synchronization, and (ii) update commitment. Figure 3.2 depicts such components. The first comprises a *replica storage subsystem*, which maintains the local data structures associated with each replica; and an *update propagation protocol*, which ensures dissemination of updates on local replicas to replicas in other sites. The second consists of a *commitment protocol*, which enforces replica consistency.[3]

In the next sections, we describe each element of the above architecture, following a logical sequence. We start, in Section 3.2.3, by isolating an individual site and describing it stores local replicas and allows applications to access them. In Section 3.2.4 we connect our site to the remainder of the distributed system, explaining how, in the absence of concurrency, remote applications may collaborate by accessing their local replicas. Finally, Section 3.2.5 addresses the possibility of concurrency and conflicts, introducing the update commitment problem.

3.2.3 Local Replica Access

At each site, the storage subsystem maintains replicas of a subset of objects. Applications may, at any time, create new objects. The creator site assigns the new object, x, a globally unique identifier, id_x, composed by a pair $id_x = \langle sid, lid \rangle$, where sid is the site identifier and lid is a locally unique identifier.

[3] Literature on optimistic replication often designates the combination of the synchronization protocol and the commitment protocol as the replication protocol.

Replicas store and manipulate *updates*. Each update results from a write request that some application issued (either locally or at a remote site).

Conceptually, a replica represents an infinite history of updates, defined by a *schedule*. We define a schedule as a totally ordered set of updates. Each update u in a schedule is marked either as *executed* (denoted $\underline{u}$) or *not-executed* (denoted $\overline{u}$). Given a schedule s, we denote by $\underline{s}$ the sub-schedule of s containing only executed updates; i.e. $\underline{s} = s \setminus \{u : \underline{u} \in s\}$. We say that a schedule s is *sound* if each update u in $\underline{s}$ is semantically compatible with the sequence of updates that precedes u in $\underline{s}$.

Each replica a maintains a local replica schedule, or simply *r-schedule*, which we denote sch_a. The r-schedule includes all updates that the replica has received. An r-schedule sch_a grows as the site learns of new updates on the object that a replicated. These may be either updates that a local application issues, or updates that some remote application issues at a remote site and that propagate through anti-entropy to the local replica (which Section 3.2.4 addresses).

The ordered execution of $\underline{sch_a}$ yields the current tentative value of a.

3.2.3.1 Implementing an r-schedule

The notion of an unlimited r-schedule is, evidently, conceptual. In practice, we need to implement an r-schedule using limited memory. A common implementation of sch_a consists of an ordered update log, complemented with a base value of a.

The log stores a limited number of updates of the r-schedule, starting from the base value. If too many updates exist in the r-schedule, then the replica prunes some of the updates from the log and applies them onto the base value. If a replica prunes updates by r-schedule-order, then the base value corresponds to the executed state obtained by executing a prefix (i.e., the pruned prefix) of the r-schedule.

Pruning must not, however, happen arbitrarily. Some updates should or must remain logged, and not immediately applied onto the base value, for several reasons. First, immediate pruning imposes whole-object propagation when synchronizing replicas, whereas update logging allows more efficient, incremental update propagation. Second, the commitment protocol may need to rollback or reorder updates that are not yet stable (see next section). Hence, we cannot discard, by pruning, the individual knowledge of such updates. Third, conflict detection regarding new updates that the system tries to add to an existing r-schedule may (depending on the system) require inspecting the semantic information associated with individual, older updates in the r-schedule (e.g. [SBK04]).

Numerous log pruning solutions have been proposed. To the best of our knowledge, none of such solutions is generic enough to be independent of the replication protocol, since all impose restrictions on the latter. Examples include Bayou's [PST+97] and Rumor's [GRR+98] log pruning solutions, each of which is tailored to the corresponding system.

Log pruning is out of the scope of the present thesis. Nevertheless, with a few adaptations, we may use the Bayou solution [PST+97]. For details, please consult

Appendix A.

Overall, Bayou prunes updates (i) in r-schedule-order, and (ii) only once they have become stable. It maintains, at each replica a, a counter (OSN_a) reflecting the number of pruned updates so far. By exchanging such a counter at the beginning of an anti-entropy session, the sender replica is able to determine: whether it may bring the receiver replica up-to-date by simply sending individual updates; or whether the sender replica needs to also transmit its base value. [4]

For simplicity of presentation in the remainder of the thesis, we will reason about r-schedules whenever we can abstract the discussion away from the implementation details.

3.2.4 Distributed Operation

So far, we have described how to provide local applications at individual sites with both read and write access to replicated objects. We now explain how local update activity disseminates across the replicated system, supporting collaboration among the distributed users.

As applications issue updates, sites need to synchronize their replicas in order to ensure freshness. Sites synchronize their replicas via an update propagation protocol, which runs during pair-wise unidirectional anti-entropy sessions. Assume that sites A and B replicate a common object, with replicas a and b, respectively. If a holds a more recent version than b (i.e. a knows of every update that b knows of, plus some new updates) then, after an anti-entropy session from A to B, b should be up-to-date with a.

A naive update propagation protocol would have the sender site, A, send the entire state of a to the receiver site, B. Obviously, such a protocol is not interesting since it will often transmit unnecessary data across the network.

A more desirable approach is to have the receiver site, B, start by sending some space-efficient representation of sch_b to the sender site, A. From such information, A can then derive the minimal set of updates that A actually needs to propagate to B in order to bring b up-to-date. Hereafter, we denote such a minimal set of updates by $\mathcal{T}$.

Algorithm 1 illustrates this approach with a protocol for anti-entropy between a sender site, A, and a receiver site, B. For the moment, we leave the space-efficient representation of sch_b unspecified. Given a replica, a, we simply assume that a space-efficient representation of sch_a exists and denote it as $[sch_a]$.

For each object that both sites replicate in common, the sender site, A, starts by comparing the schedule representations of both sites, $[sch_a]$ and $[sch_b]$, determining $\mathcal{T}$. Site A then transfers each update $u \in \mathcal{T}$ to B. As B receives each such update, B schedules u into sch_b.

The $scheduleUpdate$ function adds either $\underline{u}$ or $\overline{u}$ to some position of sch_b, ensuring that both (i) sch_b remains sound, and (ii) the stable updates in sch_b remain

[4] For safety, such a log truncation solution has assumptions on the underlying commitment protocol, as we explain in Appendix A.

Algorithm 1 Update propagation protocol from site A to site B.

1: **for all** Object x that B replicates, in replica b, **do**
2: B sends id_x and $[sch_b]$ to A.
3: **if** A replicates x, in replica a, **then**
4: A determines $\mathcal{T} = \{u : u \in sch_a \backslash [sch_b]\}$
5: **for all** Update u in sch_a such that $u \in \mathcal{T}$ **do**
6: A sends u to B
7: B sets $sch_b \leftarrow scheduleUpdate(u, sch_b)$
8: B changes $[sch_b]$ to reflect sch_b

stable (we will shortly define stability). An useful implementation of *scheduleUpdate* tries to insert $\underline{u}$ to sch_b (typically trying to append $\underline{u}$ to the end of sch_b), in order to minimize user writes that abort. In an ideal execution, updates would propagate fast enough to ensure that no remote user tries to do some subsequent work upon its replica before any previously issued updates. Here, update concurrency would not exist and *scheduleUpdate* would be able to schedule every update as executed.[5]

However, even in strongly connected scenarios, concurrency may occur. Hence, it will not always be possible to add $\underline{u}$ to sch_b, due to conflicts with the updates already in sch_b. In such a case, in order to add u, while retaining soundness of sch_b, *scheduleUpdate* is forced to add $\overline{u}$ instead. The next section will address concurrency and the consequent consistency enforcement.

3.2.5 Consistency Enforcement

Concurrent updates on distinct replicas of the same object may occur. If we only relied on the above synchronization protocol, this would lead to replicas with divergent tentative values. For instance, consider a replica a that generated some update u_1, adding it to sch_a. In the meantime, assume that another replica (of the same object), b, concurrently generates another update, u_2. Hopefully, u_1 and u_2 will be commutable. This would mean that, after two anti-entropy sessions between both replicas, both would have both updates as executed in their r-schedules. Since both updates are commutable, both schedules would be semantically equivalent (i.e. both produce semantically equivalent values of the object), and both replicas would be consistent.

However, if u_1 and u_2 happen to be incompatible, then only one of them can exist as executed in an r-schedule. Such restrictions may imply reordering or aborting updates. For instance, returning to the above example, and assuming a rea-

[5] We make the conventional assumption that every update issued at a given replica is compatible with the local r-schedule at the time the corresponding write was requested. Therefore, in the absence of concurrent updates, no conflicts may occur. Nevertheless, our contributions are still applicable should one not consider such an assumption.

sonable[6] update scheduler (*scheduleUpdate*), after issuing each update, a would have $sch_a = \underline{u_1}$, and b would have $sch_b = \underline{u_2}$. As a receives, via anti-entropy, u_2, *scheduleUpdate* will be forced to add $\overline{u_2}$ to sch_a; and vice-versa. Hence, both replicas will reach divergent values.

3.2.5.1 Explicit Eventual Consistency

We need to handle divergencies such as the previous one, by pushing the system towards a consistent state. We achieve this by complementing the synchronization protocol with a commitment protocol. Intuitively, the commitment protocol must ensure that, *eventually*, every replica adopts a schedule that is semantically equivalent to a single canonical schedule. Formally, we are interested in commitment protocols that satisfy *eventual consistency* [SS05], which we may enunciate as follows:

1. for every replica a, sch_a is sound; and
2. at any moment, for each replica a, there is a prefix[7] of sch_a, denoted $allStable_r$, that is semantically equivalent to a prefix of sch_b, for every other replica b; and
3. $allStable_r$ grows monotonically over time, by suffixing; and
4. $allStable_r$ is sound; and
5. for every update u, generated at any replica, and for every replica a, after a sufficient finite number of anti-entropy sessions, $allStable_r$ either includes $\underline{u}$ or $\overline{u}$.

Figure 3.3 illustrates eventual consistency (and other concepts that we introduce next) with an example execution of a system of three replicas (a, b and c). During this execution, each replica concurrently issues one update; a,b and c issue u_1, u_2 and u_3, respectively. The example assumes that semantics consider u_3 to be compatible and commutable with either u_1 and u_2; however, u_1 and u_2 are conflicting, hence only one of them may execute in a sound schedule. As pairs of replicas synchronize via anti-entropy (light grey arrows), they receive and schedule updates. Additionally, a commitment protocol ensures eventual consistency of the system. To abstract the example from any particular commitment protocol, we omit the operation of the commitment protocol and show only its effects on each r-schedule.

Let us restrict our attention to the evolution of sch_c and $allStable_c$ in Figure 3.3 (we will next address the remaining information in the figure). Initially (at t_1), $allStable_c$ is empty. Meanwhile, sch_c changes gradually. Clearly, every sch_c is

[6] I.e., one that does not abort updates when all updates to schedule are compatible.

[7] We consider the usual reflexive notion of prefix (and suffix); i.e. a schedule is a prefix (and a suffix) of itself.

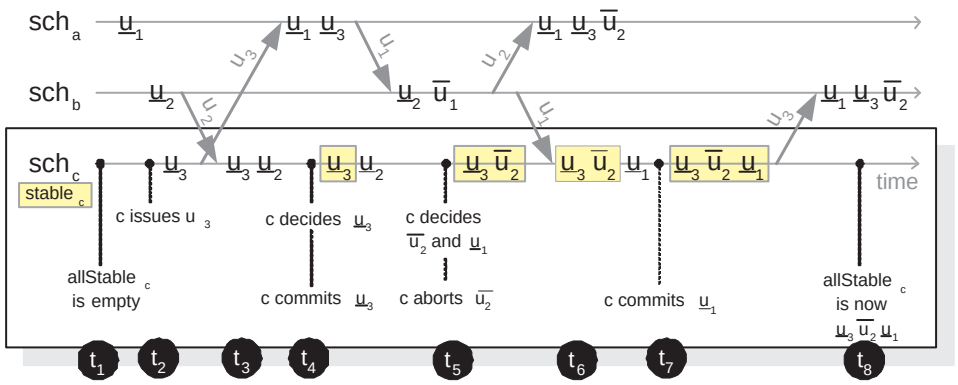


Figure 3.3: Example execution of a system ensuring explicit eventual consistency (EEC), from the point of view of applications at the site with replica *c*. The largest stable prefix ($stable_c$) of sch_c is identified, at each point in time, by a shaded box.

sound. Sometimes, the evolution of sch_c is not monotonic;[8] for instance, at t_5, when sch_c changes from $\underline{u_3 u_2}$ to $\underline{u_3}\overline{u_2}$. In contrast, $allStable_c$ does evolve monotonically (by suffixing). Finally, at time t_8, $allStable_c$ becomes $\underline{u_3}\overline{u_2}\underline{u_1}$. Such a prefix of sch_c is guaranteed to be semantically equivalent to a prefix of sch_a and sch_b, which is true: both sch_a and sch_b are $\underline{u_1 u_3}\overline{u_2}$, which is equivalent to $\underline{u_3}\overline{u_2}\underline{u_1}$ since u_1 and u_3 are commutable.

We use the terms *decided* and *stable* distinctly to characterize updates and schedules.[9] Intuitively, decision is only concerned with agreement, while stability implies both agreement and local reception of the update(s) upon which agreement was reached. We say that a given schedule s is *decided* at a replica a if, based on the local state of a, the protocol guarantees that s will eventually be a prefix of $allStable_r$. Furthermore, when s is decided and a prefix of sch_a (i.e., s is locally available at sch_a), we say that s is *stable* at a.

Given an update u such that there exists at least one schedule containing u that is decided (resp. stable) at some replica r, we say u is decided (resp. stable) at a. Finally, we say that an update for which one such schedule does not exist (hence, the update is not yet decided at a) is *tentative* at a.

We further differentiate between decided and stable updates depending on whether they appear executed or not-executed in the corresponding schedule. If $\underline{u}$ is decided at a, we say that u is *decided for commitment* at a. Further, if $\underline{u}$ is stable at a, we say that u is *committed* at a. In contrast, such a distinction is irrelevant if the decision (at some replica a) is $\overline{u}$; we always designate u as *aborted*. In practice, an update need not be available at some replica a for a to abort it. Therefore, once some

[8] Note that eventual consistency requires $allStable_c$ to evolve monotonically by suffixing, but not sch_c.

[9] Although in most related literature both terms are synonyms, this distinction is necessary for some parts of the thesis.

update u becomes decided as $\overline{u}$ at some replica a, we implicitly consider u as stable at a, even if the actual update is not available at a.

We are especially interested in commitment protocols that both:

1. Satisfy eventual consistency;
   and

2. Are able to explicitly detect, anytime, whether an update $\underline{u}$ (or $\overline{u}$) in sch_a, is either tentative or stable.

We call such requirements *explicit eventual consistency* (EEC). Given some replica a, $stable_a$ denotes the largest prefix of sch_a that is stable at a. We call $stable_a$ the *stable r-schedule* of a. The ordered execution of the committed updates in $stable_a$ (i.e. $\underline{stable_a}$) produces the committed value of a. As $stable_a$ grows with new committed updates, their execution is guaranteed to never rollback.

We illustrate EEC by revisiting the example in Figure 3.3. Focusing on replica c, Figure 3.3 depicts each commitment decision that c takes. $stable_c$ starts (at t_1) as an empty schedule (similarly to sch_c). sch_c grows as c schedules updates u_3 (at t_2) and u_2 (at t_3). Updates u_3 and u_2, being tentative at time t_3, are not initially present in $stable_c$. At time t_4, c learns that the commitment protocol has decided $\underline{u_3}$.[10]Since c holds u_3 (u_3 is in sch_c), c commits $\underline{u_3}$; hence, $stable_c$ includes $\underline{u_3}$ at t_4. Later, at t_5, c learns that the commitment protocol has agreed on $\underline{u_1}$ and $\overline{u_2}$. Accordingly, c aborts u_2, making $stable_c = \underline{u_3}\overline{u_2}$. Since u_1 is not yet available at sch_c, c decides but does not commit $\underline{u_1}$. Only when c receives a from some remote replica (at t_6) will c commit $\underline{u_1}$, causing $stable_c$ to become $\underline{u_3}\overline{u_2}\underline{u_1}$.[11]

We believe EEC is not only desirable but fundamental for most collaborative applications. EEC allows both non-critical and critical tasks of an application to use the same replicated system. The former may access the tentative value and profit from the high availability of optimistic replication. On the other hand, the latter may impose stronger consistency demands by accessing the committed value, at the cost of availability. More precisely, critical tasks that need to read data that is guaranteed to never rollback will access the committed value. Critical tasks that need to write and demand the same consistency guarantees as in a pessimistic replication system may achieve that by (i) writing to the tentative value and (ii) waiting until the corresponding updates become stable. Such critical tasks will then have access to a value that ensures the strong consistency criterion of sequential consistency [Lam79], a common guarantee of pessimistic replicated systems. A number of representative systems offer EEC, such as Bayou [PST+97], IceCube[KRSD01], Coda [KS91], CVS [C+93], and Deno [Kel99].

[10]By some means that do not concern us for the moment.

[11]This example exposes the difference between $stable_c$ and $allStable_c$. The former consists of a schedule that the commitment protocol guarantees that will eventually be a prefix of the latter. A system that ensures EEC must be able to expose the former to applications. The later, in contrast, is merely conceptual and need not be determined by the system.

3.2.5.2 Abstract Commitment Protocol

This section describes an abstract commitment protocol ensuring EEC. The input of the protocol comprises the updates that each replica receives via anti-entropy. Conceptually, we distinguish two main steps in the commitment protocol, which occur in parallel to new update activity:

1. Proposing a candidate, where a replica proposes a candidate schedule to become stable.

2. Agreement outcome, where a replica becomes aware of agreement on a larger stable prefix of r-schedule.

Some distributed agreement algorithm achieves the above agreement. At this stage, we leave the agreement algorithm as a black box. Algorithm 2 outlines a generic, abstract commitment protocol.

Algorithm 2 Update Commitment at replica *a* (not assumed to be atomic)

```
1: loop
2:    if new sch_a exists then
3:       proposeSch(sch_a);
4: ||
5: loop
6:    if sch_a ≠ stable_a(sch_a) then
7:       stable_a ← getNewDecidedSch(sch_a);
8:       sch_a ← scheduleUpdates(sch_a \ stable_a, stable_a);
```

Algorithm 2 abstracts the two commitment steps, which run at each replica, by the *proposeSch* and *getNewDecidedSch* functions. The former proposes the local r-schedule as a candidate for agreement, eventually the replica schedules new tentative updates. From such a candidate proposal, and possibly from other candidates that other replicas propose concurrently, the agreement protocol reaches consensus on a single *canonical* stable schedule, possibly incrementally by successively growing prefixes.

The r-schedule at every site eventually converges to become equivalent to the canonical stable schedule, as its replica both learns of new agreement outcomes (i.e. new decided updates) and receives the corresponding decided updates. At each convergence step, $stable_a$ grows to become a larger suffix of sch_a. Note that $stable_a$ (and, inherently, sch_a) may converge to a schedule with a different update order than the canonical schedule's order, as long as equivalence between both holds.

The *getNewDecidedSch* function blocks until the local replica, *a*, learns of a larger stable prefix of sch_a, returning it so that the replica may adopt it as its new stable r-schedule ($stable_a$). In other words, each time *getNewDecidedSch* returns, one or more tentative updates in sch_a have become decided. Such decisions may

imply readjustments to sch_a. First, updates may be decided (by the agreement algorithm) in a different position in sch_a and in a different state (either $\underline{u}$ or $\overline{u}$) than when they were tentatively scheduled. Second, updates that come after the decided updates (i.e. updates in $sch_a \backslash stable_a$) may have to be rearranged in order to ensure the soundness of the new sch_a, if a different stable prefix exists. Such a rearrangement may involve reordering updates, changing updates from an executed to an not-executed state (i.e. rolling-back updates), and vice-versa.

Recall Figure 3.3. At time t_4, c blocks on *getNewDecidedSch*. At t_5, *getNewDecidedSch* returns $\overline{u_3}\underline{u_2}$. This causes sch_c to change from $\overline{u_3 u_2}$ to $\overline{u_3}\underline{u_2}$ (thus becoming the same as $stable_c$).

3.2.6 Bayou-Like Baseline Solution

The previous description of an optimistic replication system is still open in a number of aspects, leaving space for many different concrete solutions. This section presents one particular solution, a variation of Bayou [PST+97].[12] We build on the Bayou approach because it is representative of a large class of existing systems (as we describe in Chapter 2).

We do not claim any novelty in this particular solution. Instead, it lays the ground for the novel contributions that the remainder of the thesis introduces. Hereafter, we call it the baseline solution. From the baseline solution, we later depart to each contribution, modifying and extending some of the components of the baseline solution, with the goal of minimizing or eliminating most of its main shortcomings.

3.2.6.1 Update Propagation

We start by addressing update propagation. In order to instantiate our update propagation protocol (Algorithm 1, Section 3.2.4), we need to have a space-efficient representation of an r-schedule (denoted $[sch_a]$ for some replica a). In the baseline solution, we use version vectors [PPR+83] as such a representation. A version vector consists of a vector of counters, one per site replicating the corresponding object. In order to use version vectors, we impose two constraints on the update scheduler (*scheduleUpdate* and *scheduleUpdates* functions in the previous algorithms) and on the update propagation protocol. Namely:

1. Update scheduling preserves local update order.

   The order at which each replica issues updates constitutes a partial order among all updates in the replicated system. We require that the r-schedules that *scheduleUpdate* and *scheduleUpdates* construct always satisfy this partial order.

[12] The variation includes minor terminology differences, restructuration of some algorithmic parts for a simplified presentation, and an optimization of the update propagation protocol when udpate truncation is supported.

For such a purpose, we assign a version identifier to each update, similarly to object identifiers (see Section 3.2.3). A version identifier consists of a pair $\langle sid, lid \rangle$, where sid is the identifier of the site that issued the update and lid is a monotonically-increasing local counter. We require that, for any pair of updates, u_1 and u_2, scheduled in an r-schedule, if $u_1.sid = u_2.sid$ and $u_1.lid < u_2.lid$, then u_1 is necessarily ordered before u_2 in the r-schedule.

2. Update propagation ensures the *prefix property* [PST+97].

   The prefix property means that if the r-schedule of some replica a holds some update with $sid = i$ and $lid = x$, the r-schedule will also hold all updates with $sid = i$ and $lid < x$. The update propagation protocol must propagate updates in an order that ensures the prefix property (shortly, we explain how).

Given the r-schedule of some replica, a, at site A, the entry of $[sch_a]$ corresponding to some site B (denoted $[sch_a][B]$) has the highest lid from the updates in sch_a that site B issued (i.e. those with $sid = i$). Since an r-schedule is consistent with local update order and the prefix property holds, such a value corresponds to the lid of the most recent update in the r-schedule with $sid = i$. It can be shown that, given the above two constraints, a version vector holds sufficient information to represent an r-schedule. In particular, one can use version vectors to compare two r-schedules, in order to determine which new updates one r-schedule includes that the other r-schedule does not.

Before describing an update propagation protocol using version vectors, we must introduce some basic notions. Let x and y be two version vectors. Hereafter, we say that y *dominates* x, or $x \leq y$, to denote that $\forall i, x[i] \leq y[i]$; and say that y *strictly dominates* x, or $x < y$, to denote that $x \leq y$ and $\exists j : x[j] < y[j]$). Further, we say that x and y are concurrent, or $x \parallel y$, when neither $x \leq y$ nor $y \leq x$. Finally, given x and y, their merging is represented by a version vector, denoted $merge(x,y)$, such that $\forall i, merge(x,y)[i] = max(x[i], y[i])$.

Using the version vector representation of r-schedules, we may now easily instantiate the update propagation protocol, as Algorithm 3 shows.

Algorithm 3 Update propagation protocol from site A to site B.

```
1:  for all Object x that B replicates, with replica b do
2:    B sends id_x and [sch_b] to A.
3:    if A replicates x, with replica a then
4:      Let u be the first update in sch_a.
5:      while u exists do
6:        if [sch_b][u.sid] < u.lid then
7:          A sends u to B.
8:          B sets sch_b ← scheduleUpdate(u, sch_b)
9:          B sets [sch_b][u.sid] ← u.lid
10:       Let u be the next update in sch_a.
```

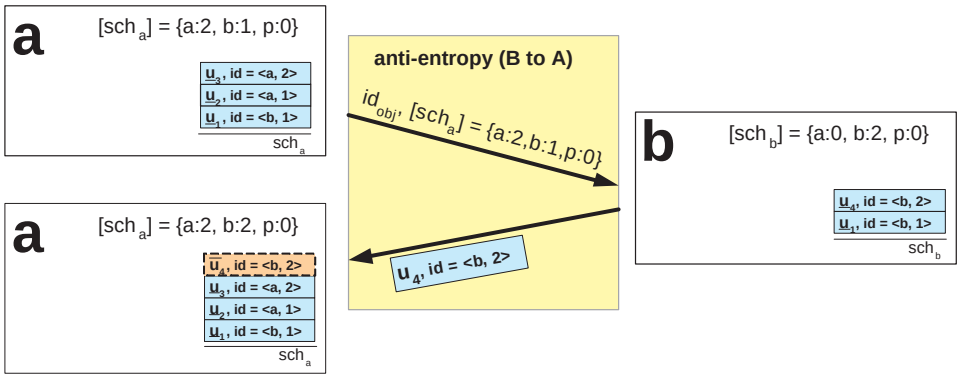


Figure 3.4: Example of update propagation using the baseline solution, in a system with an object replicated across three sites, *A*, *B* and *P* (at replicas *a*, *b* and *p*, respectively). In the example, site *A* synchronizes from *B*.

The minimal set of updates to transfer (set $\mathcal{T}$ in Algorithm 1) now includes the updates in the sender's r-schedule that have a higher *lid* than the *lid* of the corresponding entry in the version vector of the receiver replica (line 6). In order to ensure the prefix property, updates must now propagate in the order by which they appear in the sender's r-schedule; hence, $\mathcal{T}$ is now an ordered set.

Figure 3.4 depicts an example of the above protocol, in a system with an object replicated across three sites, *A*, *B* and *P* (at replicas *a*, *b* and *p*, respectively). In the example, replica *a* starts with $sch_a = \underline{u_1}\underline{u_2}\underline{u_3}$, while *b* has $sch_b = \underline{u_1}\underline{u_4}$. Site *A* initiates an update propagation session from *B*, sending the version vector representing the r-schedule for the only object *A* replicates. Having received such a version vector (along with the object identifier), *B* is able to determine that *b* has one update (u_4) that *a* does not yet hold (since $[sch_a][b] < 2$), and hence transmits it. After receiving u_4, *a* tries to schedule $\underline{u_4}$ in sch_a, necessarily after $\underline{u_1}$, in order to preserve local update order. However, *a* detects that, due to semantics, no such safe scheduling exists and thus adds $\overline{u_4}$ to sch_a.

At the end, *a* and *b* have a mutually inconsistent tentative view of the object. The next section describes an update commitment protocol, which enforces convergence towards consistency.

3.2.6.2 Update Commitment

The baseline solution adopts the simple, primary commit approach for update commitment. In this approach, a single, distinguished primary replica decides its own r-schedule as the canonical stable schedule.

In primary commit, the *proposeSch* function (see Algorithm 2), is defined by Algorithm 4. Line 2 means that, in the case of the primary replica, any new updates in sch_a are immediately decided (line 2); otherwise, no decision is taken.

To briefly illustrate the primary commitment protocol, we return to the system

Algorithm 4 *proposeSch*(*schedules*) at replica *a*

1: **if** replica *a* is primary **then**
2: $stable_a \leftarrow sch_a$
3: **else**
4: Do nothing.

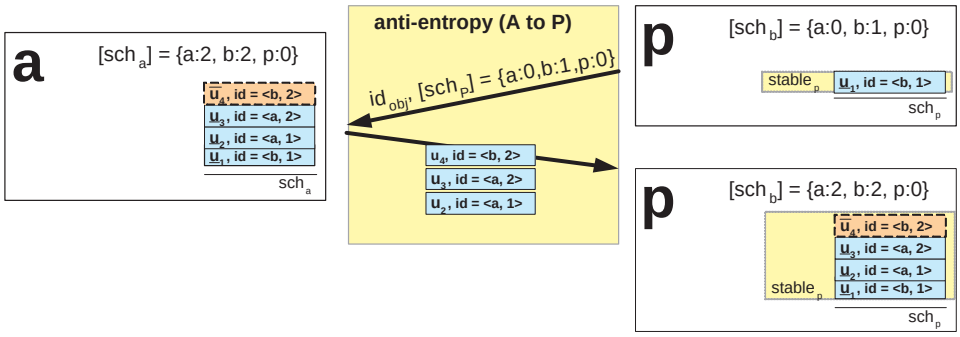


Figure 3.5: Example of update propagation using the baseline solution.

of the previous example. Some time after the previous example, site *P*, holding the primary replica of the object (replica *p*), initiates an update propagation session from site *A*. Figure 3.5 depicts such a session. By analysis of $[sch_p]$, *a* determines the minimal set of updates to send to *p*. After receiving such updates, *p* schedules them at sch_p and, by Algorithm 4, immediately determines that such a schedule is stable.

Having obtained a larger stable r-schedule at the primary replica, the commitment protocol disseminates this information to the remaining replicas. Decision information explicitly includes, for each newly decided update, *u*: (i) the identifier of *u* and (ii) its state (either $\overline{u}$ or $\underline{u}$) at the primary's stable r-schedule. Implicitly, decision information also specifies, for each decided update, its order within the primary's stable r-schedule, as we explain shortly.

Decision information flows through the same anti-entropy sessions by which we propagate tentative updates. As each replica *a* learns of decisions it was not aware of, *a* applies them, incrementing the local $stable_a$ with newly decided updates that correspond to the new decision information. As a result, each replica incrementally builds a stable r-schedule that grows to eventually become identical[13] to the stable r-schedule of the primary replica.

The protocol for decision information propagation is part of the commitment

[13]The primary commitment protocol ensures a stronger consistency guarantee than EEC, which we call totally-ordered EEC. Essentially, totally-ordered EEC complements EEC with the constraint that, more than equivalent, $allStable_r$ must be *identical* to a prefix of sch_b, for every replica *b*. It is easy to show that, since $allStable_r$ grows monotonically by suffixing, the commitment protocol is, in fact, deciding a total order among the stable updates.

protocol. Algorithm 5 depicts the decision information propagation protocol from a sender replica, a, to a receiver replica, b. For efficiency, the sender replica sends only decision information concerning the stable update identifiers in $stable_a$ that are not yet stable at b. Determining such a minimal set of updates is easy since the primary commitment protocol ensures that all stable updates are totally ordered. Therefore, it suffices for both replicas to compare the number of stable updates at each replica in order to determine which decided updates the sender must actually propagate. Given some replica a, the terminology of [PST+97] denotes such a number of stable updates by CSN_a.

Algorithm 5 Decision information propagation from replica a to replica b, during anti-entropy from sites A to B

1: B sends CSN_b and $[sch_b]$ to A.
2: **if** $CSN_b < CSN_a$ **then**
3: **for all** update u in $stable_a$, starting at the one after the position corresponding to CSN_b, and following r-schedule order **do**
4: **if** $[sch_b][u.sid] \geq u.lid$ **then**
5: A sends the identifier of u and its executed state ($\underline{u}$ or $\overline{u}$) to B
6: B looks u up in sch_b
7: At B, *getNewDecidedSch* either returns $stable_b;\underline{u}$ or $stable_b;\overline{u}$ (depending on the state received from A)
8: **else**
9: Break

Essentially, the protocol starts by determining whether a has more stable updates than b, comparing CSN_b with CSN_a (line 2). If so (i.e. $CSN_b < CSN_a$), then a, for the updates that b holds (i.e., those in $[sch_b]$), informs b about the corresponding decision that the commitment protocol took (line 5). Such a description comprises the update identifier and a flag indicating whether the update was decided for commitment or abort. The sender replica notifies each decision in (total) commitment order. The receiver replica simply looks up the (previously tentative) update corresponding to each identifier it receives and constructs a new stable r-schedule, which *getNewDecidedSch* then returns (line 7). By Algorithm 2, such a new stable r-schedule will then affect the current state of the replica.

We present an example of decision information propagation in Figure 3.6. Recall the previous example, where the primary replica, p, had determined a stable schedule $\underline{u_1 u_2 u_3 \overline{u_4}}$. Assume that site P had propagated such decision information to A, and now site B initiates an anti-entropy session from A. As a first step, an update propagation session is carried out, where replica b schedules two previously unknown updates (u_2 and u_3) in sch_b. Since b detects conflicts between such updates and its current r-schedule, b schedules both as not-executed.

As a second step, A propagates decision information to B. Such a step starts by having B send the $[sch_b]$ version vector and the number of stable updates b holds, SCN_b; in this moment, b has not yet heard of any decision and thus holds no stable

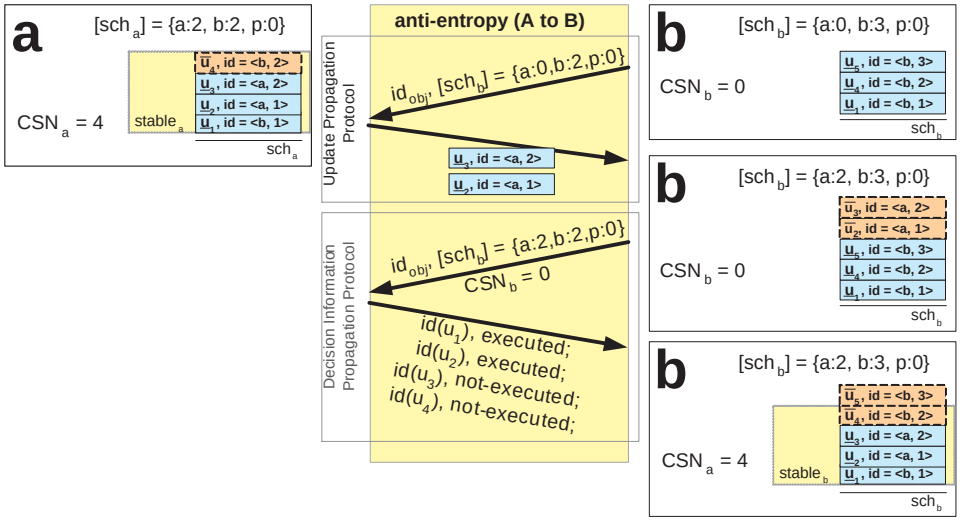


Figure 3.6: Example of decision information propagation using the baseline solution.

updates. As a result, A notifies B of the decisions that a is aware of concerning updates that b holds (in sch_b). Since such notifications follow schedule order, b applies them in that order to sch_b. Due to the changes to the stable prefix of sch_b, other (tentative) updates in sch_b may be reordered or rolled-back. In this example, u_5 is rolled-back due to conflicts with the new stable r-schedule.

3.3 Overview of Our Contributions

The baseline solution constructed so far has crucial limitations when we consider the three key requirements of optimistic replication, as Chapter 1 enunciates: rapid update commitment, abort minimization and adaptation to network and memory constraints. The baseline solution exhibits the following main limitations, among others:

1. Commitment protocol depends on a single point of failure.

   Should the primary replica fail or become temporarily inaccessible due to network partitioning, the commitment protocol halts. Therefore, in the presence of weak connectivity, the dependence on a centralized site may severely compromise the goal of rapid update commitment.

2. Update propagation and commitment protocols restrict the allowed schedule space.

   The baseline solution restricts the tentative schedule space (by imposing that schedules satisfy the local update order) and on the committed schedule space (by further imposing totally ordered EEC). Such restrictions limit

the universe of solutions available to each replica scheduler, hence limiting its ability to minimize aborts. Moreover, totally ordered EEC implies agreement on a total order among stable updates, whereas EEC would only require a weaker agreement (e.g. on a partial order). Agreement on a total order typically requires exchanging more messages between replicas than weaker forms of agreement (e.g. [PS02, Lam05]), hence slowing down update commitment.

3. Replica synchronization evolves only via replica-to-replica interactions.

   For some object, both the update propagation protocol and the commitment protocol evolve exclusively via anti-entropy sessions between pairs of replicas of that object. Other opportunities of interaction, namely between arbitrary sites, no matter whether they replicate the object or not, are neglected. In weakly connected scenarios, opportunities for replica-to-replica anti-entropy with acceptable bandwidth may be relatively rare when compared with the opportunities for interaction between arbitrary pairs of sites. Hence, by limiting replica synchronization to replica-to-replica interactions, the baseline solution hinders the progress of update propagation and, consequently, of rapid update commitment.

4. Replicas store and propagate whole, potentially large updates.

   The baseline solution tries to reduce memory and storage usage with update data by enabling update pruning, and by determining the minimal set of updates (and, possibly, base value) to propagate during anti-entropy. In the baseline solution, replicas store and propagate updates and base values in a plain form. If we consider cases where updates have non-negligible sizes, their plain storage and propagation may easily become prohibitive in either memory-constrained devices and/or bandwidth-constrained networks, respectively.

The remainder of the present thesis describes novel contributions that eliminate or minimize each of the above shortcomings, either by extension or by modification of components of the baseline solution. Figure 3.7 revisits the baseline component diagram (see Figure 3.2), illustrating the components from the baseline solution that our contributions extend or modify. Our contributions comprise: (i) decentralized commitment protocols that replace the primary commit protocol; (ii) an extension of commitment agreement algorithms that allows their progress via exchange of consistency packets between arbitrary sites, including non-replicas; and (iii) a data deduplication scheme for reducing effective memory and network usage with updates and base values.

The next sections give an intuitive overview of such contributions, before the following chapters plunge into a precise presentation of each contribution.

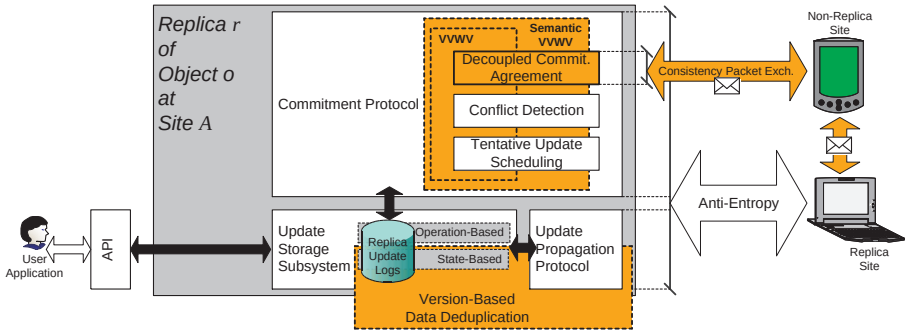


Figure 3.7: Functional components of the replication protocol at each site and their interactions with applications, non-replica sites and other sites, considering the contributions of this thesis.

3.3.1 Commitment Protocols

We propose commitment protocols that are decentralized and rely on epidemic quorum systems [Kel99, HSAA03] for agreement.

3.3.1.1 Novel Results on Epidemic Quorum Systems

Epidemic quorum systems allow reliable agreement in weakly connected environments [Kel99, HSAA03]. By requiring only subsets of replicas, called quorums and anti-quorums, to vote for a proposed schedule to decide it, epidemic quorum systems avoid the single point of failure of the primary commit approach. Hence, the unavailability of a particular replica does not necessarily prohibit the progress of the update commitment process. Most importantly, epidemic quorum systems can decide even when quorums and anti-quorums are never simultaneously accessible: voting information flows epidemically between replicas and each replica learns of decisions based solely on its local state.

Taking such advantages into account, literature proposing epidemic quorum systems [Kel99, HSAA03] argues that they are adequate to weakly connected networks, where connected quorums and anti-quorums may be improbable. Accordingly, we base our commitment protocols on epidemic quorum systems.

However, apart from some the proposed epidemic quorum systems, no proper study, either formal or empirical, has been devoted to analyzing their trade-offs and comparing the alternatives. Hence, before devising commitment protocols that rely on epidemic quorum systems, we formally define and study the availability and performance of existing alternatives. We obtain novel results that help us better understand epidemic quorum systems, which we rely on when devising the actual commitment protocols. In particular, our study shows evidence that plurality-based quorum systems have higher availability and performance than majority-based ones.

We address this contribution in Section 4.1 of Chapter 4.

3.3.1.2 Commitment Protocol

When no explicit semantic information about the updates is available, we use a novel commitment protocol, called Version Vector Weighted Voting Protocol (VVWV). VVWV relates updates by the corresponding happens-before relation [Lam78], relying on version vectors. VVWV follows a conservative approach: it considers any concurrent updates (by happens-before) as conflicting (i.e. only one update among a set of mutually concurrent updates may be executed in a sound update schedule). Therefore, it always aborts some updates when concurrency occurs.

VVWV offers efficient and reliable update commitment through the use of an epidemic quorum system. For ease of management of the dynamic replica set, VVWV relies on epidemic weighted voting for defining quorums and antiquorums [Kel99]. VVWV uses a plurality scheme for higher availability and performance. The results we have obtained from the previously mentioned analytical study justify such a choice.

VVWV regards schedules containing concurrent tentative updates as rival candidates in an election. Each replica of a given logical object acts as a voter whose votes collectively determine the outcome of each election. A candidate schedule wins an election by collecting a plurality of votes, in which case it commits and its rival candidates abort.

In optimistic replication, applications may often generate more than one tentative update prior to their commitment decision. Since the commitment decision may not be taken in the short-term, these applications will then issue a sequence of multiple, *happens-before-ordered* tentative updates. Basic epidemic weighted voting solutions [Kel99] are not efficient in such a scenario, since can only commit one update per election round. VVWV introduces a significant improvement by allowing multiple update candidates to commit in a single election round.

VVWV represents multiple update candidates by the version vector corresponding to the tentative version obtained if the entire update sequence was applied to the replica. The voting algorithm relies on the expressiveness of version vectors to decide whether the whole update sequence, or a prefix of it, should commit. Consequently, candidates consisting of one or more happens-before-related updates may be committed on a single distributed election round. In weakly connected network environments, where such update patterns are expectably dominant, a substantial reduction of the update commitment delay is therefore achievable; not only relatively to basic weighted voting solutions, but also, under certain conditions, to the primary commit solution. We support such claims with experimental results from simulations. In worst case scenarios, VVWV behaves similarly to basic weighted voting protocols.

We address this contribution in Section 4.2 of Chapter 4.

Complementarily, joint work with Pierre Sutra and Marc Shapiro has generalized VVWV in order to take advantage of a rich semantic repertoire to minimize aborted work and further accelerate commitment. We describe and discuss the

protocol resulting from such work in Section 4.3 of Chapter 4.

3.3.2 Decoupled Commitment by Consistency Packet Exchange

Considering a given object, anti-entropy sessions restrict progress of the update propagation and commitment protocols to cases where pairs of replicas of the object become accessible. It neglects opportunities of interaction with a possibly dense universe of non-replicas. Offline anti-entropy (see Section 2.8) would be a first solution to exploit the presence of such non-replicas. However, it entails relatively high memory requirements on non-replicas when replicated objects are sufficiently large (in state-based systems [SS05]), or writes are frequent (in operation-transfer systems [SS05]). In these cases, such requirements may be prohibitive for a significant amount of non-replicas, especially in memory-constrained ubiquitous environments; thus, the presence of such non-replicas would still be largely neglected.

In alternative to offline anti-entropy, we propose to complement anti-entropy sessions by the exchange of consistency packets between replicas and non-replicas, and among non-replicas.[14] A consistency packet includes lightweight consistency meta-data, which is meaningful for the agreement algorithm of the commitment protocol (concerning some replicated object). When non-replicas are within reach, a replica produces a consistency packet and delivers it to the former. Non-replicas buffer consistency packets, delivering them to other sites (either replicas or non-replicas, in reference to the object associated with each consistency packet) they may run across. Therefore, non-replicas act as temporary, unreliable carriers of commitment agreement meta-data, comprising a transport channel that supplements the standard replica-to-replica anti-entropy sessions. This way, the agreement algorithm may also progress as non-replicas carry consistency packets across replicas.

The lightweight property of consistency packets allows even non-replicas with significant resource constraints to carry and deliver consistency packets. As our current real world reaches closer to Weiser's vision of ubiquitous computing [Wei91], the set of (possibly resource constrained) non-replicas surrounding any particular group of replicas becomes denser; thus, their contribution to consistency packet exchange becomes more significant.

We need to modify a commitment protocol in order to effectively take advantage of individual consistency packets being disseminated, decoupled from the corresponding update data. We achieve that by decoupling of the agreement algorithm from the remaining phases of a commitment protocol. We propose such a decoupling for a generic commitment protocol. Further, we apply it to VVWV.

Figure 3.8 illustrates the new system architecture, when extended to exploit the presence of non-replicas as carriers of consistency packets. We show that the simple exchange of consistency packetsby non-replicas may accelerate commitment,

[14] Note, however, the system may still exploit the (expectedly more rare) resource-rich non-replicas with offline anti-entropy.

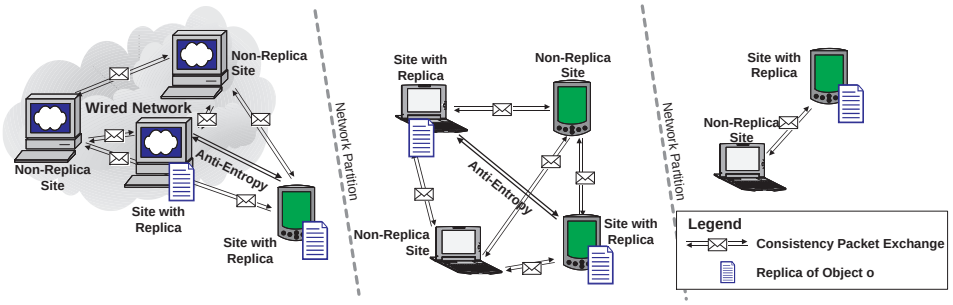


Figure 3.8: Illustration of system architecture, extended with consistency packet exchange via non-replicas. Mobile and stationary sites replicate object *A* and interact by pair-wise unidirectional anti-entropy. Non-replica sites co-exist with the replicated system, buffering and delivering consistency packets.

reduce aborts, and allow more network-efficient update propagation. We support such a statement with simulation results.

We address this contribution in Chapter 5.

3.3.3 Versioning-Based Deduplication

We propose a novel scheme for efficient replica storage and synchronization when state-based updates are employed. In contrast to operation-based updates, which are typically space- and network-efficient [LLS99], state-based updates impose significant storage and communication overheads for large objects.

We exploit similarities that may exist across updates (and between updates and base values), either belonging to the same object, or across different objects. This technique is called data deduplication, and may transparently be combined with conventional techniques such as data compression or caching. It works by determining redundant chunks of data, across the updates/base values that some site stores, or across the updates/base values stored by two sites that wish to propagate updates to one another. In the former case, the local site needs not store the redundant chunks (but one *root* chunk); when necessary, we may always obtain the contents of the redundant chunks from the root chunk. In the latter case, the sender site needs not transmit the chunks that are redundant across it and the receiver site; the receiver site may simply obtain the redundant chunks locally instead of downloading them from the network.

The key challenge of data deduplication lies in detecting redundant chunks, most notably in the update propagation situation. Our central contribution consists of a method for distributed chunk redundancy detection that relies on the exchange of version vectors representing the version state of sites. When compared with other recent prominent solutions for distributed data deduplication [MCM01,

CN02, BF04, JDT05, ELW+07, BJD06], which rely on the compare-by-hash technique [Hen03], our contribution has important advantages. First, we rely on deterministic similarity detection, instead of the probabilistic compare-by-hash technique, hence avoiding the possibility of data corruption due to hash collisions. Second, we allow faster distributed deduplication than compare-by-hash, with a lower network overhead, while attaining (at least) comparable redundancy detection efficiency. Finally, our solution is also designed with concerns for efficient storage and local data access.

Intuitively, our scheme starts by having a receiver site, b, exchanging the $[sch_b]$ and $[pruned_b]$ version vectors of each of its replicas. By comparison with their own version vectors, the sender site may determine a set of updates/base values that both sites currently hold in common. Before sending each chunk of data of some update or base value during anti-entropy, the sender checks whether such a chunk is also present in any of the common updates/base values; if so, then the chunk is redundant and the protocol avoids transferring it. By maintaining adequate data structures, we can perform the latter check efficiently and in a deterministic way.

Overall, our solution achieves significant reductions on memory usage, network consumption and synchronization performance. Hence, we contribute to adapt the optimistic replication service to the resources of more constrained sites and networks. We support such claims with experimental results obtained with real workloads. We show that, with real workloads, our scheme: (1) achieves lower transferred content and meta-data volumes than all evaluated compare-by-hash solutions; and (2), with few exceptions, synchronizes replicas faster than all evaluated alternatives.

We address this contribution in Chapter 6.

3.4 Summary

We propose an abstract optimistic replication middleware service for support of collaborative applications in weakly connected resource-constrained environments. It is designed as a replication service for generic objects, supporting both non-semantic, state-based and semantic, operation-based write requests. A commitment protocol ensures explicit eventual consistency (EEC); intuitively, this means that, eventually, every replica converges, by monotonic steps, to an equivalent state and that each such step is explicitly visible to applications. We explain why an adequate instantiation of the described abstract replication service may satisfactorily support a wide range of collaborative applications, with differing semantics and consistency demands.

Borrowing techniques from representative state-of-the-art systems, we then construct a baseline instantiation. It constitutes a baseline solution with important disadvantages, which are representative of many state-of-the-art systems. It serves as a common base, from which each contribution of the thesis departs, modifying

and extending some of components of the baseline solution in order to minimize or eliminate a relevant subset of the latter's shortcomings.

We summarize our contributions, mapping them against the components of the baseline solution that each one modifies or extends. Namely, we give the intuition behind: the commitment protocols; decoupled commitment agreement by consistency packet exchange; and our scheme for efficient and corruption-free duplicate elimination for update storage and propagation. The next chapters pursue these directions, describing and evaluating each contribution in detail.

Chapter 4

Update Commitment

This chapter presents the commitment protocol of our replication service. The commitment protocol is called called Version Vector Weighted Voting Protocol (VVWV). VVWV relies on an epidemic quorum system, in order to tolerate the partitioning that one expects frequent in the environments that the thesis considers.

We organize the remainder of the chapter as follows. We start, in Section 4.1 by studying epidemic quorum systems, as a generic means of achieving fault-tolerant agreement in distributed systems. We obtain novel results, which help better understand the availability and performance of epidemic quorum systems.

Epidemic quorum systems can be used to directly solve the update commitment problem. Nevertheless, we observe that such a direct solution has the crucial drawback of requiring one election round per decided update, which becomes increasingly expensive as connectivity weakens.

Driven by such an observation and by the above novel results, we present VVWV. VVWV leverages the previous solution by an efficient analysis of multiple-update candidates that allows deciding and committing multiple updates in a single election round. Underlying such an analysis is the use of version vectors as an efficient and expressive means for representing and reasoning about multiple-update candidates.

Finally, remarking that VVWV is semantic-oblivious, we set out to describe an extension to VVWV so as to support rich semantics, in Section 4.3. It consists of an abstract extension of VVWV that is able to incorporate the repertoire of a representative semantic framework. It is not a complete solution, as it lacks an adequate definition of *eligible candidates*, which we leave open. We propose and discuss possible definitions, exposing why they are not entirely satisfactory.

4.1 Update Commitment by Epidemic Quorum Algorithms

Quorum systems are a basic tool for reliable agreement in distributed systems [PW95]. Their applicability is wide, ranging from distributed mutual exclusion, name servers, selective dissemination of data, to distributed access control and

signatures [AW96]. In particular, update commitment protocols can also rely on quorum systems for highly available data replication.

Classical quorum systems require agreement on a value to be accepted by a *quorum* of live sites that are simultaneously connected in the same network partition. This is not adequate in weakly connected networks, e.g. mobile or sensor networks, where connected quorums are improbable.

Epidemic quorum systems eliminate such a shortcoming, allowing unconnected quorums. An epidemic quorum algorithm tries to ensure agreement by running a finite number of *elections*. Intuitively, in each election, each site may vote for a single proposed value. By epidemic propagation of votes, eventually each site should be able to determine, from its local state, whether the system has agreed on a given value w, or the current election is inconclusive and, hence, a new one starts.

Recent work [Kel99, HSAA03] proposed epidemic quorum algorithms for update commitment. However, to the best of our knowledge, neither are epidemic quorum systems well defined, nor are their availability or performance well studied. In addition, the extensive studies on classical quorum systems are not valid for epidemic quorum systems, as their failure conditions and quorum definitions are radically distinct.

Taking into account the weakly connected environments addressed herein, the commitment protocol we propose relies on epidemic quorum systems. Therefore, it is fundamental to guide the design choices of the protocol by strong results concerning epidemic quorum systems and algorithms.

This section studies epidemic quorum systems and algorithms, unveiling novel results that help understand such an approach. Such a study will then serve as the basis for the commitment protocols that we propose in the following chapters. Our contribution is threefold:

1. We formally define epidemic coteries and a generic epidemic quorum algorithm, and prove its safety. The formalism unifies existing epidemic quorum systems and is a framework for devising and studying epidemic coteries.

2. Based on the formalism, we characterize formally the liveness of epidemic quorum systems. Such a characterization enables a novel, analytical comparison of the availability and performance of epidemic quorum systems. Further, we present conditions where epidemic quorum systems decide faster than classical ones, and identify trade-offs that do not exist in the classic approach.

3. We compare the availability and performance of two state-of-the-art epidemic coteries, majority and linear plurality. To the best of our knowledge, this constitutes the first comparative evaluation between the two epidemic coteries.

We start by formally defining epidemic quorum systems and algorithms, presenting some fundamental properties, in Section 4.1.1. We then proceed, in Section 4.1.2, to formally study their liveness, both concerning their availability and

performance. Finally, we focus our attention on two existing epidemic coteries, comparing their theoretical availability and performance in Appendix 4.1.3.

We have proven all the results presented in the next sections. When absent from the text, the proof of each theorem/lemma/proposition may be found at the end of the Chapter, in Section B.2. We should also mention that the results we present next have been successfully validated by a brute-force execution generator[1], up to $z = 4$ and $y = 4$ (we define these parameters shortly).[2]

4.1.1 Epidemic Quorum Systems

Consider that the set of distributed sites, denoted *Sites*, wishes to agree on a single value, taken from a set of values, denoted *Val*, each of which was proposed by some site. In the case of a commitment protocol, the values correspond to conflicting schedules that compete for commitment. Accordingly, $z = |Val|$ measures contention: $z = 0$ and $z = 1$ correspond to the trivial scenarios of no contention, while $z > 1$ imply some contention between two or more concurrently proposed values.

We approach the above problem using epidemic quorum algorithms. Similarly to classical quorum algorithms [PW95], epidemic quorum algorithms ensure agreement on a single value by having coteries of sites vote for the proposed values. However, the epidemic and classical notions of coteries are distinct, as well as the corresponding algorithms.

4.1.1.1 Epidemic Coteries

We start with some definitions and lemmas that will allow us to construct and reason about epidemic quorum systems in the remainder of the paper.

Definition 1. *Vote set, vote configuration and value-vote configuration*

A vote set v is a set of sites, $v \subseteq Sites$.

A vote configuration c is a set of non-empty vote sets, $c = \{v_1^c, v_2^c, .., v_n^c\}$ such that, for all v_i^c and v_j^c in c, $v_i^c \cap v_j^c = \emptyset$.

A value-vote configuration vc is a set of pairs, $vc = \{\langle a_1, v_1^{vc}\rangle, .., \langle a_n, v_n^{vc}\rangle\}$, where:

- *each $a_1, .., a_n \in Val$,*
- *for all i and j, $a_i \neq a_j$,*
- *and the set formed by $\{v_1^{vc}, .., v_n^{vc}\}$ is a vote configuration.*

[1] For a given set of system parameters (such as *y* and *z*), the generator simulates *every* possible execution of the algorithms addressed. The obtained results are then compared against the ones obtained by the analytical expressions.

[2] Due to the exponential growth of brute-force execution generation time, the time to process higher values of *z* and *y* is unacceptable (order of magnitude of hours, or higher).

We will use vote and value-vote configurations to reason about the votes that each site is aware of in a given moment.

To simplify our presentation when handling value-vote configurations, we introduce the following two auxiliary notations. Given some value-vote configuration $vc = \{\langle\ a_1, v_1^{vc}\rangle, .., \langle\ a_n, v_n^{vc}\rangle\}$, we denote the vote configuration obtained by $\{v_1^{vc}, .., v_n^{vc}\}$ as $VCfg(vc)$. Furthermore, for each value a_i in vc, we denote the corresponding vote set (i.e. v_i^{vc}) as $votes_{vc}(a_i)$.

Hereafter, when reasoning about sets A and B, we use the notation $A \backslash B$ to denote the set difference between A and B.

When considering agreement on a given value, we will also be interested in differentiating the votes for such a value from the other votes. For this reason, we introduce the definition of q-vote configuration.

Definition 2. *Q-vote configuration.*

A q-vote configuration qc is a pair, $qc = \langle Q^{qc}, \{A_1^{qc}, A_2^{qc}, .., A_n^{qc}\}\rangle$,
where $\{Q^{qc}, A_1^{qc}, .., A_n^{qc}\}$ *is a vote configuration.*

We call set Q^{qc} the *quorum* of q-vote configuration *qc*, and each set A_i^{qc} as an *anti-quorum*[3] of *qc*. Intuitively, we will use q-vote configurations when reasoning about a particular distribution of votes, either real or hypothetical, with respect to some value being considered. The quorum represents the set of voters for the considered value, while each anti-quorum represents the different sets of voters for each rival value.

Deciding election outcomes will require speculating about the votes that some value may potentially receive in the best case; i.e. if all sites whose vote is still unknown by the local site happen to vote for that value. The potential vote set reflects such a notion.

Definition 3. *Potential vote set*

Let v be a vote set. The potential vote set of v in a vote configuration c, denoted $[v]_c$, *is defined as* $[v]_c = v \cup (Sites \backslash (\bigcup\ v_i : v_i \in\ c)\}$.

In particular, $[\emptyset]_c = (Sites \backslash (\bigcup\ v_i : v_i \in\ c)\}$.

Further, the potential vote set of s in a q-vote configuration qc, denoted $[s]_{qc}$, *is defined as* $[s]_{qc} = s \cup\ (Sites \backslash\ (Q^{qc} \cup A_1^{qc} \cup\ .. \cup A_{n_{qc}}^{qc}))$.

We are now able to define the notions of coverage and potential coverage of a q-vote configuration by another one, the building blocks for a definition of epidemic coterie. We say that a larger q-vote configuration, *qc*, covers a smaller one, *qd*, ($qc \succ\ qd$) when each vote set in the latter is a subset of a distinct vote set of the former, and Q^{qc} is a superset of Q^{qd}. Similarly, potential coverage ($qc \rhd\ qd$) means that a q-vote configuration may still grow (with the unknown votes) to cover another one.

Definition 4. *Coverage of a q-vote configuration.*

[3] The expression is due to Holliday et al [HSAA03].

Let $qc = \langle Q^{qc}, \{A_1^{qc}, A_2^{qc}, .., A_n^{qc}\}\rangle$ *and* $qd = \langle Q^{qd}, \{A_1^{qd}, A_2^{qd}, .., A_m^{qd}\}\rangle$ *be q-vote configurations. We say that qc* covers *qd, or* $qc \succ qd$*, if and only if:*

1. $Q^{qc} \supseteq Q^{qd}$*, and*
2. *There exists an injective function* $f : \{1,..,m\} \rightarrow \{1,..,n\}$ *such that, for all* $1 \leq k \leq m$, $A_{f(k)}^{qc} \supseteq A_k^{qd}$.

Definition 5. *Potential coverage of a q-vote configuration.*

Let $qc = \langle Q^{qc}, \{A_1^{qc}, A_2^{qc}, .., A_n^{qc}\}\rangle$ *and* $qd = \langle Q^{qd}, \{A_1^{qd}, A_2^{qd}, .., A_m^{qd}\}\rangle$ *be q-vote configurations. We say that qc* may cover *qd, or* $qc \rhd qd$*, if and only if:*

1. $[Q^c]_{qc} \supseteq Q^{qd}$*, and*
2. *There exists an injective function* $f : \{1,..,m\} \rightarrow \{1,..,n\}$ *such that, for all* $1 \leq k \leq m$, $[A_{f(k)}^{qc}]_{qc} \supseteq A_k^{qd}$ *or* $[\emptyset]_{qc} \supseteq A_k^{qd}$.

Otherwise, we say that qc cannot cover *qd, which we denote* $qc \not\rhd qd$.

We illustrate coverage and potential coverage with a simple example. Consider a system with five sites, $Sites = \{A, B, C, D, E\}$. Let qc and qd be q-vote configurations such that:

$$qc = \langle \{A\}, \{C\}\}\rangle$$

$$qd = \langle \{A, E\}, \{\{B\}, \{D\}\}\rangle$$

It is easy to see that qc does not cover qd; in fact, the quorum of qc is a subset of the quorum of qd, and qc has no anti-quorum that is a superset of the anti-quorum $\{D\}$ of qd. However, it is true that qc may cover qd. Intuitively, we may observe this by noting that it is possible to add the missing sites (B, D and E) to qc in such a way that the resulting q-vote configuration will cover qd. One possibility is to add E to the quorum, and B and D to the first and second anti-quorums of qc, thus obtaining $qc' = \langle \{A, E\}, \{\{B\}, \{C, D\}\}\rangle$.

We may finally define epidemic coteries (EC).

Definition 6. *Epidemic Coterie*

Let $\mathcal{E}$ *be a non-empty set of q-vote configurations.* $\mathcal{E}$ *is an* Epidemic Coterie (EC) *if,* $\forall qc, qd \in \mathcal{E} : qc \neq qd$*:*

1. $\forall j : \langle A_j^{qc}, \{Q^c\} \cup \{A_i^{qc} : i \neq j\}\rangle \not\rhd qd$*, and*
2. $\langle \emptyset, \{Q^c\} \cup \{A_1^{qc}, .., A_n^{qc}\}\rangle \not\rhd qd$*, and*
3. $qc \not\succ qd$.

We note $QSet(\mathcal{E})$ the set of quorums of $\mathcal{E}$, i.e. the set of every $Q^{qc} : qc \in \mathcal{E}$; and by $AQSet(\mathcal{E})$ the set of anti-quorums of $\mathcal{E}$, i.e. the set of every $A_j^{qc} : qc \in \mathcal{E}$, for all j.

Intuitively, the above definition ensures the following. Consider that, given a set of votes, we may form a q-vote configuration whose quorum votes for a value *w* and which covers a q-vote configuration in the EC. Then, no arbitrary assignment of the missing votes may reach another vote configuration whose quorum votes for a value other than *w*. (Note that other vote configurations may be reached, as long as their quorums vote for *w*.) This will ensure the safety of an epidemic quorum algorithm to to decide *w*, as we will show shortly.

4.1.1.2 Examples: Majority and Linear Plurality ECs

This section defines two relevant examples of ECs, $\mathcal{E}_{Maj}$ and $\mathcal{LP}$, in a system with five sites, $Sites = \{A,B,C,D,E\}$. $\mathcal{E}_{Maj}$ and $\mathcal{LP}$ are instances of majority and linear plurality constructions, respectively; such constructions are subjacent to the systems proposed and empirically studied by Holliday et al. [HSAA03] and Keleher [Kel99].[4]

The Majority EC, $\mathcal{E}_{Maj}$, is the set of q-vote configurations such that $QSet(\mathcal{E}_{Maj})$ is the collection of all sets of 3 sites (i.e. $(|Sites|+1)/2$ in an odd sized system), and $AQSet(\mathcal{E}_{Maj})$ is the empty set. The absence of any anti-quorum from $\mathcal{E}_{Maj}$ is explained by observing that the majority of votes for some value implies that no other value may obtain a majority from the other votes, *independently* of how such votes are distributed. Hence, anti-quorums are not needed to define $\mathcal{E}_{Maj}$.

Definition 7. *Majority EC*

$$\mathcal{E}_{Maj} = \{\langle q, \emptyset\rangle : q \subseteq Sites \wedge\ |q| = 3\}$$

The linear plurality EC, $\mathcal{LP}$, is a superset of $\mathcal{E}_{Maj}$. We define $\mathcal{LP}$ as follows:

Definition 8. *Linear plurality EC*

$$\mathcal{LP} \quad = \quad \left\{ qc : \left\{ \begin{array}{l} \forall k, |Q^{qc}| > |[\emptyset]_{qc}| \text{ or } (|Q^{qc}| = |[\emptyset]_{qc}| \wedge Q^{qc} <_{break} [\emptyset]_{qc} \\ \text{and} \\ \forall k, |Q^{qc}| > |[A_k^{qc}]_{qc}| \text{ or } (|Q^{qc}| = |[A_k^{qc}]_{qc}| \wedge Q^{qc} <_{break} [A_k^{qc}]_{qc} \\ \text{and} \\ \forall qd \in \mathcal{LP}, qc \not> qd \end{array} \right. \right\}$$

where $<_{break}$ is the tie-breaking relation between vote sets. Formally, $<_{break}$ is defined as follows: given two vote sets, v and v':

$v <_{break} v'$ if and only if $\exists i \in v : \forall l \in v', i < l$.

[4] It should be noted that they do not present the mentioned ECs using our EC formalism.

Quorum	Anti-Quorums
p_1, p_2, p_3	
p_1, p_2, p_4	
p_1, p_2, p_5	
p_1, p_3, p_4	
p_1, p_3, p_5	
p_1, p_4, p_5	
p_2, p_3, p_4	
p_2, p_3, p_5	
p_2, p_4, p_5	
p_3, p_4, p_5	

Quorum	Anti-Quorums
p_1, p_2	$\{p_3\}, \{p_4\}$
p_1, p_2	$\{p_3\}, \{p_5\}$
p_1, p_2	$\{p_4\}, \{p_5\}$
p_1, p_3	$\{p_2\}, \{p_4\}$
p_1, p_3	$\{p_2\}, \{p_5\}$
p_1, p_3	$\{p_4\}, \{p_5\}$
p_1, p_4	$\{p_2\}, \{p_3\}$
p_1, p_4	$\{p_2\}, \{p_5\}$
p_1, p_4	$\{p_3\}, \{p_5\}$
p_1, p_5	$\{p_2\}, \{p_3\}$
p_1, p_5	$\{p_2\}, \{p_4\}$
p_1, p_5	$\{p_3\}, \{p_4\}$

Quorum	Anti-Quorums
p_2, p_3	$\{p_1\}, \{p_4, p_5\}$
p_2, p_3	$\{p_1\}, \{p_4\}, \{, p_5\}$
p_2, p_4	$\{p_1\}, \{p_3, p_5\}$
p_2, p_4	$\{p_1\}, \{p_3\}, \{, p_5\}$
p_2, p_5	$\{p_1\}, \{p_3, p_4\}$
p_2, p_5	$\{p_1\}, \{p_3\}, \{, p_4\}$
p_3, p_4	$\{p_1\}, \{p_2\}, \{, p_5\}$
p_3, p_5	$\{p_1\}, \{p_2\}, \{, p_4\}$
p_4, p_5	$\{p_1\}, \{p_2\}, \{, p_3\}$
p_1	$\{p_2\}, \{p_3\}, \{, p_4\}, \{, p_5\}$

Figure 4.1: Q-vote configurations comprising $\mathcal{LP}$, as obtained from Definition 12. $\mathcal{E}_{Maj}$ is a subset of $\mathcal{LP}$, and corresponds to the left-most table only.

Intuitively, given two candidate values, the tie-breaking relation chooses in favor of the value that is voted for by a site that has a lower identifier than any site that may potentially vote for the other value.

In contrast to $\mathcal{E}_{Maj}$, some q-vote configurations of $\mathcal{LP}$ include anti-quorums. These correspond to non-majority cases, where the election of some value depends on how the remaining votes are distributed among the other values.

The plurality condition requires that, in order to decide a value *v*, the number of voters for *v* (the quorum) must be higher than the size of the potential vote set of voters for every all other value (the anti-quorums). Linear plurality assumes a total order among sites, and weakens the previous plurality condition.

In addition to q-vote configurations that respect plurality, linear plurality also allows q-vote configurations whose quorum is equally sized as some anti-quorums in the same q-vote configuration, as long as the lowest sites of the potential vote sets of the former are higher than that of the latter. This is ensured by the tie-breaking relation.

Figure 4.1 presents the q-vote configurations that respectively comprise $\mathcal{E}_{Maj}$ and $\mathcal{LP}$ in a system with 5 processes. Naturally, $\mathcal{E}_{Maj}$ is a subset of $\mathcal{LP}$. As the reader may confirm, both $\mathcal{E}_{Maj}$ and $\mathcal{LP}$ are ECs.

4.1.1.3 Epidemic Quorum Algorithm

This section presents an algorithm that relies on a given EC, $\mathcal{E}$, for achieving distributed consensus, and proves its safety. We start by defining the local state the algorithm maintains at each site. Let $A \in$ *Sites* be a site. The local state of A includes:

1. A *current election identifier*, $e_A \in N$;
2. A value-vote configuration, $V_A = \{\langle a_1, v_1^A \rangle, .., \langle a_n, v_n^A \rangle\}$.

We call each $votes_{V_A}(a_i)$ the *local vote set* at A for value a_i. We designate the vote configuration $VCfg(V_A)$ as the *local vote configuration* at A. For simplicity of presentation, we use $votes_A(i)$ to denote $votes_{V_A}(i)$; further, when there does not exist any value-vote pair for some value i in V_A, $votes_A(i)$ denotes the empty set, ø.

Having introduced the state, we now describe the epidemic quorum algorithm, which Algorithm 6 outlines.

Algorithm 6 Epidemic Quorum Algorithm (not assumed atomic)

```
1:  loop  // Value Proposal
2:     Choose a value v ∈ Val
3:     propose(v)
4:  ||
5:  loop  // Vote Propagation
6:     Choose an accessible site B
7:     receiveVotesFrom(B)
8:  ||
9:  loop  // Wait For Election Outcome
10:    checkOutcome()
```

The state of each site A evolves as A proposes a value, receives voting information from other sites, and determines the outcome of each election. Algorithms 8, 9 and 10 describe each of such steps, respectively. For presentation simplicity, we impose that each such algorithm runs atomically; nevertheless, the algorithms may be broken down in finer atomic grains. Algorithm 7 is an auxiliary function that Algorithms 8 and 9 call.

Algorithm 7 merge(value-vote configuration V_r)

```
1: for all ⟨ a_i, v_i^r ⟩ ∈ V_r do
2:    if votes_A(a_i) ≠ ø then
3:       votes_A(a_i) ← votes_A(a_i) ∪ v_i^r
4:    else
5:       V_p ← V_p ∪ ⟨ a_i, v_i^r ⟩
```

Algorithm 8 propose(value v)

```
1: if ∀i : p ∉ v_i^A then
2:    merge({⟨ v, {p}⟩})
3: else
4:    merge({⟨ v, ø⟩})
```

When a value v is proposed at site A (Algorithm 8), A votes for it if A has not yet voted for another value in the current election; this means adding A to the vote set corresponding to v in V_p (line 2). For fairness, the algorithm ensures that, even

if A has already voted in the current election, V_p includes an entry for v, possibly with an empty vote set (line 4); this way, sites other than A may still learn of v and vote for it.

Algorithm 9 receiveVotesFrom(site B)

```
1: if e_A < e_B then
2:     e_A ← e_B
3:     for all ⟨ a_i, v_i^A ⟩ ∈ V_p do
4:         v_j^A ← ø
5: if e_A = e_B then
6:     merge(V_r)
7:     if ∀v ∈ VCfg(V_p), p ∉ v then
8:         choose some j : 1 ≤ j ≤ |V_p|
9:         propose(a_j)
```

When A is accessible from some site B, A may receive voting information from B, as Algorithm 9 describes. If B already holds information about a later election, then A discards its obsolete vote information (in V_p) and sets e_A to the more recent election, e_B (lines 1 to 4).

Finally, if both sites end up in the same election (this does not happen if $e_B < e_A$), then A complements V_p with new votes learned from B (line 6). Further, in case A has not yet voted in its current election, it may cast a vote for one of the newly received values (lines 7 to 9).

Algorithm 10 checkOutcome()

```
1: if repeat-condition then
2:     V_p ← ø
3:     e_A ← e_A + 1
4: if decide-condition(w) then
5:     decide w
```

Based exclusively on its local state, each site determines if an election is guaranteed to produce the same outcome at every other site. Two possible outcomes are possible: the election may either decide one of the proposed values (*decide*), or not decide any value (*repeat*). In the latter case, a new election starts; this is repeated until some election decides, as Algorithm 10 describes.

The conditions for each outcome are based on the EC the algorithm uses. They are the following.

Definition 9. decide-condition(w) *and* repeat-condition
Let $p \in Sites$ be a site in an EC $\mathcal{E}$.

- decide-condition(w) *at A is defined as:*

  $\exists w \in Val : \exists qd \in \mathcal{E} : \langle votes_A(w), \{votes_A(j) : j \neq w\} \rangle \succ qd.$

- repeat-condition *at A is defined as:*

  $\forall w \in Val : \nexists qd \in \mathcal{E} : \langle votes_A(w), \{votes_A(j) : j \neq w\}\rangle \rhd qd.$

Intuitively, each local vote set at A, $votes_A(i)$, grows monotonically as votes are cast and propagated. Ideally, the local vote sets should grow so that eventually they can form a q-vote configuration that covers one q-vote configuration in the EC. When that happens, the value w associated with the quorum of such a q-vote configuration (i.e. $votes_A(w)$ is the quorum in such a q-vote configuration) will be decided at A.

Otherwise, while *decide-condition(w)* is not verified yet, one may determine if the local vote sets may still grow to form a q-vote configuration that will cover some q-vote configuration in the EC. If that does not hold for any q-vote configuration in the EC, then the current election round will never decide any value, and a new one may then safely start; in this case, each $votes_A(i)$ is emptied and its monotonic growth restarts.

As an example, recall the ECs $\mathcal{E}_{Maj}$ and $\mathcal{LP}$, introduced in Section 4.1.1.1. Consider that A has the following local vote sets: $votes_A(a) = \{B,D\}$, $votes_A(b) = \{C\}$ and $votes_A(c) = \{E\}$. At this moment, neither *decide-condition*(a), nor *decide-condition*(b), nor *decide-condition*(c), nor *repeat-condition* are verified (neither for $\mathcal{E}_{Maj}$, nor for $\mathcal{LP}$).

Hence, A waits for further votes. Consider that A later learns that A has voted for c (i.e. $votes_A(c) = \{A,E\}$). If we consider $\mathcal{E}_{Maj}$, then clearly the local vote-sets of A *cannot cover* any q-vote configuration in $\mathcal{E}_{Maj}$; hence, *repeat-condition* is true and a new election starts.

Let us, otherwise, consider $\mathcal{LP}$. In this case, A decides, since the votes that A knows of form a q-vote configuration that covers some q-vote configuration in the $\mathcal{LP}$ EC. It is trivial to show that one such q-vote configuration is $d = \langle\{A,E\},\{\{B\},\{C\}\}\rangle$, since $\langle votes_A(c), \{votes_A(a), votes_A(b)\}\rangle$ (which corresponds to $\langle\{A,E\},\{\{B,D\},\{C\}\}\rangle$) covers d. Since the quorum of the local q-vote configuration is the set of voters for value c, A decides value c.

Naturally, it is crucial that, having A decided c, any other site will also decide the same value. The definition of EC ensures the safety of the values that the system agrees on, according to the requirements of classical consensus [Lam05]. The next lemmas are useful to prove such a statement.

Lemma 1. *Let qc and qd be q-vote configurations such that* $qd \succ qc$. *For any* A_i^{qd}, *either there exists* A_j^{qc} *such that* $[A_j^{qc}]_{qc} \supseteq [A_i^{qd}]_{qd}$, *or* $[\emptyset]_{qc} \supseteq [A_i^{qd}]_{qd}$.

Lemma 2. *Let A and B be two sites where* $e_A = e_B$. *Then,* $votes_A(x) \subseteq [votes_B(x)]_{c_r}$ *for all* $x \in Val$.

We can finally prove that previous algorithms are safe.

Theorem 1. *The epidemic quorum algorithm satisfies the following requirements:*

1. *Any value decided must have been proposed at some site (nontriviality).*

2. *A site can only decide a single value (stability).*

3. *Two different sites cannot decide different values (consistency).*

Proof. Clearly, the algorithm ensures nontriviality and stability.

To prove consistency, assume, for purposes of contradiction, that two sites, A and B, decide different values a and b (resp.), at elections e_A and e_B (resp.). Let us first assume that $e_A = e_B$. By Definition 9, we know that, when A decides, there exists a q-vote configuration, qd in $\mathcal{E}$, such that $\langle votes_A(a), \{votes_A(j) : j \neq a\}\rangle \succ qd$. So, by Lemma 1, for any value i, there either exists some A_j^{qd} such that $[A_j^{qd}]_{qd} \supseteq [votes_A(a_i)]_{V_p}$, or $[\emptyset]_{qd} \supseteq [votes_A(a_i)]_{V_p}$ (1). Since $\mathcal{E}$ is an EC, definition 6 tells us that, for any $qc \in \mathcal{E}$ ($qc \neq qd$), $\forall j : \langle A_j^{qd}, \{Q^{qd}\} \cup \{A_i^{qd} : i \neq j\}\rangle \not\succ qc$, and $\langle \emptyset, \{Q^{qd}\} \cup \{A_1^{qd}, .., A_n^{qd}\}\rangle \not\succ qc$ (2). From (1) and (2), we know that $\langle votes_A(b), \{votes_A(j) : j \neq b\}\rangle \not\succ qc$ (3).

However, the outcome of the same election at B was *decide* for value b. Therefore, we know that, at the time of such a decision, there exists some $qc \in \mathcal{E}$ such that $\langle votes_B(b), \{votes_B(j) : j \neq b\}\rangle \succ qc$. Then, by Lemma 2, for any value i, $votes_B(i) \subseteq [votes_A(i)]_{V_A}$.

Hence, $\langle votes_A(b), \{votes_A(j) : j \neq b\}\rangle \rhd \langle votes_B(b), \{votes_B(j) : j \neq b\}\rangle$, and, transitively, $\langle votes_A(b), \{votes_A(j) : j \neq b\}\rangle \rhd qc$, which contradicts (3).

Let us now assume that $e_A < e_B$ (or vice-versa). Clearly, at least one site C (possibly the same as B) must have determined the outcome of election e_A as *repeat*; otherwise, B would not have reached e_B. In such a moment, by Def. 9, $\nexists qc \in \mathcal{E} : \langle [votes_C(b)]_{V_C}, \{[votes_C(j)]_{V_C} : j \neq b\}\rangle \succ qc$. However, since the outcome of the same election at A was *decide*, $\exists qd \in \mathcal{E} : \langle votes_A(b), \{votes_A(j) : j \neq b\}\rangle \succ qc$. It follows from Lemma 2 that, for all i, $votes_A(i) \subseteq [votes_C(b)]_{V_C}$. Hence, $\langle [votes_C(b)]_{V_c}, \{[votes_C(j)]_{V_C} : j \neq b\}\rangle \succ \langle votes_A(b), \{votes_A(j) : j \neq b\}\rangle$, and, transitively, $\langle [votes_C(b)]_{V_C}, \{[votes_C(j)]_{V_C} : j \neq b\}\rangle \succ qc$, which is a contradiction. □

4.1.2 Liveness of Epidemic Quorum Systems

We characterize the liveness of an EC by its availability and the time to decide. Since we deal with a distributed system, measures are obtained at a particular, arbitrary site A; e.g. *availability* means the *availability seen by arbitrary site A*. Also, we start measuring the time A takes to decide at the moment when A first votes for some arbitrary value.

We divide time into *communication rounds* (or simply *rounds*). We assume a round to be an upper bound on the time needed for A to send and, subsequently, receive vote information to/from every other correct (i.e. not faulty) site in its current partition. Further, we assume communication is live between sites simultaneously in the same partition and partition changes occur only at the end of each communication round.

We use a simple probabilistic failure model where the probability of permanent, non-byzantine failures is constant and uniform; we denote it by p_f. For simplicity, we assume the probability that a correct site votes in a value in *Val* to be uniform, $\frac{1}{z}$. Finally, we assume eventual vote propagation between correct sites; i.e., in spite of transient partitioning, A is able to eventually send and receive vote information to/from any other correct site.

Given a particular EC, $\mathcal{E}$, we characterize it by two probability distribution functions, namely $dec_{\mathcal{E}}(n)$ and $rep_{\mathcal{E}}(n)$. Value $dec_{\mathcal{E}}(n)$ (resp. $rep_{\mathcal{E}}(n)$) denotes the (conditional) probability that A, having collected n votes in a given election, evaluates *decide-condition(w)* for some value w (resp. *repeat-condition*) as true. More precisely:

Definition 10. *$dec_{\mathcal{E}}$ and $rep_{\mathcal{E}}$*

Let $T_{all}(n)$ be the collection of value-vote configurations such that, for each $vc_i \in T_{all}(n)$, $\sum_{s \in VCfg(vc_i)} |s| = n$. Given an EC $\mathcal{E}$, let $T_d^{\mathcal{E}}(n)$ and $T_r^{\mathcal{E}}(n)$ be:

$$T_d^{\mathcal{E}}(n) = \{c : c \in T_{all}(n) \text{ and } \exists s \in VCfg(c) : \exists d \in \mathcal{E} : \langle s, \{t : t \in (VCfg(c) \backslash \{s\})\}\rangle \succ d\}$$
$$T_r^{\mathcal{E}}(n) = \{c : c \in T_{all}(n) \text{ and } \forall s \in VCfg(c) : \nexists d \in \mathcal{E} : \langle s, \{t : t \in (VCfg(c) \backslash \{s\})\}\rangle \rhd d\}$$

We define $dec_{\mathcal{E}}$ and $rep_{\mathcal{E}}$ as: $dec_{\mathcal{E}}(n) = |T_d^{\mathcal{E}}(n)|/z^n$ and $rep_{\mathcal{E}}(n) = |T_r^{\mathcal{E}}(n)|/z^n$.

Clearly, $rep_{\mathcal{E}}(n)$ and $dec_{\mathcal{E}}(n)$ correspond to exclusive events. Hereafter, we restrict our analysis to ECs where $\forall n : rep_{\mathcal{E}}(n) < 1$.[5]

We are able to characterize availability and number of rounds to decide in any generic EC, as long as we know its $dec_{\mathcal{E}}(n)$ and $rep_{\mathcal{E}}(n)$. Such probability distributions enable us to abstract our study from the details of each particular EC.

Hereafter, we write $\binom{x}{y}$ to denote the binomial coefficient of x and y.

4.1.2.1 Availability

We adopt the conventional definition of availability [PW95] of quorum systems, according to which availability is the probability that the system will eventually agree on some value. Since we assume eventual vote propagation between correct sites, transient partitioning is irrelevant when quantifying eventual decision; thus, we look at the system of correct sites as co-existing in the same (imaginary) partition.

Agreement is achieved by a possibly null, finite sequence of elections with a *repeat* outcome, followed by one election with a *decide* outcome. Therefore, the probability that the system decides is given by the probability that it either decides in the first election; or that the first election has *repeat* outcome and the second election decides; and so forth. We precisely derive the availability of an EC quantifying such a probability, as follows.

[5] This means that we leave out useless ECs that can only infinitely repeat elections, and never decide.

Theorem 2. *The availability of EC $\mathcal{E}$ is given by:*

$$\sum_{n=0}^{|Sites|} \frac{\left(\begin{array}{c} y \\ n \end{array} \right) (1-p_f)^n \; p_f^{(|Sites|-n)} dec_{\mathcal{E}}(n)}{1-rep_{\mathcal{E}}(n)} \tag{4.1}$$

Due to [FLP85], availability is less than one if at least one site may be faulty. Looking at the above expression, this implies $\forall n, dec_{\mathcal{E}}(n) + rep_{\mathcal{E}}(n) < 1$, which means that an election may block (i.e. never return any outcome).

4.1.2.2 Communication Rounds To Decide

Availability determines the probability that a system decides within a finite period of time, no matter how long it is.

we are also interested in knowing how fast it reaches such a decision. We may characterize that by the probability of decision within a given number of rounds, r. We denote such a probability distribution as $P_d^{\mathcal{E}}(r)$.

In order to derive $P_d^{\mathcal{E}}(r)$, we need to obtain the probability that A collects new votes at each round. $p_v(v_i \rightarrow v_f; r)$ denotes the probability that, starting from v_i votes, A collects v_f votes after r rounds. p_v depends on how one models network inaccessibility due to transient partition changes. However, independently of such a model, we know that $p_v(v_i \rightarrow v_f; r) = 0$ if $v_f < v_i$; it is trivial to show that the algorithm ensures such a property.

Different models of network inaccessibility may be considered. Their study is out of the scope of the present thesis. As an illustration, one possible model of network inaccessibility is the following: at the end of a given communication round, the partition A belongs to does not include a given *correct* site with a constant and uniform probability, h. We designate such a model as $\mathcal{M}_p$. We precisely quantify (and prove correct) the probability distribution p_v with model $\mathcal{M}_p$ in Appendix B.2.

We may finally obtain the probability of decision within a given number of rounds.

Theorem 3. *Probability of decision within r rounds*

The probability that an arbitrary site, A, starting with $V_p = \emptyset$, decides any value within r or less rounds, $P_d^{\mathcal{E}}(r)$, is given by:

$$P_d^{\mathcal{E}}(r) = \begin{cases} 0, & \text{if} \quad r = 0 \\ \sum_{v_r=0}^{|Sites|} p_v(0 \rightarrow v_r; r) dec_{\mathcal{E}}(v_r) + f(0,r), & \text{if} \quad r > 0 \end{cases} \tag{4.2}$$

where f is defined as:

$$f(v_i, r) = \sum_{v_x=v_i}^{|Sites|} p_v(v_i \rightarrow v_x; 1)[(rep_{\mathcal{E}}(v_x) - rep_{\mathcal{E}}(v_i)) P_d^{\mathcal{E}}(r-1) + f(v_x, r-1)] \tag{4.3}$$

n=	0	1	2	3	4	5	
$dec_{Plur}(n)$	0	0	0	0,1111	0,4815	1	Plurality (3 proposals)
$rep_{Plur}(n)$	0	0	0	0	0	0	
$dec_{Maj}(n)$	0	0	0	0,1111	0,3333	0,6296	Majority (3 proposals)
$rep_{Maj}(n)$	0	0	0	0	0	0,3704	
n=	0	1	2	3	4	5	
$dec_{Plur}(n)$	0	0	0	0,0625	0,3906	1	Plurality (4 proposals)
$rep_{Plur}(n)$	0	0	0	0	0	0	
$dec_{Maj}(n)$	0	0	0	0,0625	0,2031	0,4141	Majority (4 proposals)
$rep_{Maj}(n)$	0	0	0	0	0,0938	0,5859	
	0	1	2	3	4	5	
$dec_{Plur}(n)$	0	0	0	0,04	0,328	1	Plurality (5 proposals)
$rep_{Plur}(n)$	0	0	0	0	0	0	
$dec_{Maj}(n)$	0	0	0	0,04	0,136	0,2896	Majority (5 proposals)
$rep_{Maj}(n)$	0	0	0	0	0,192	0,7104	

Figure 4.2: Values of $dec(n)$ and $rep(n)$ for $\mathcal{E}_{Maj}$ and $\mathcal{LP}$ in a system with five sites. We consider the cases where 3, 4 and 5 values are proposed ($z = 3..5$); we omit the cases of 0, 1 and 2 values proposed, since both $\mathcal{E}_{Maj}$ and $\mathcal{LP}$ behave identically in such cases.

Clearly, $P_d^{\mathcal{E}}(0)$ is zero since *decide-condition* is impossible with an empty V_p. With $r > 0$, one may distinguish two situations that fulfill *decide-condition*. A first one corresponds to the cases where, after r or less rounds, A collected enough votes, v_r, to decide in a single election; this situation corresponds to the first term of the equation with $r > 0$ in (4.2). The second term of such an equation corresponds to the cases where at least one *repeat-condition* is verified, and followed by a *decide*. A detailed description of both terms is complex, and can be found in Appendix B.2.

The above theorem unveils an interesting trade-off between availability and performance that is intrinsic to ECs. Such a trade-off will become clear in the next section, which illustrates it by studying the liveness the majority ($\mathcal{E}_{Maj}$) and linear plurality ($\mathcal{LP}$) ECs.

4.1.3 Liveness Study of Majority and Linear Plurality ECs

This section studies the liveness characteristics of the majority ($\mathcal{E}_{Maj}$) and linear plurality ($\mathcal{LP}$) ECs (as Section 4.1.1.2 defines) in the light of the previous definitions. The obtained results are the first formal results concerning the state-of-the-art systems [Kel99, HSAA03] that rely on such ECs.

From $\mathcal{E}_{Maj}$ and $\mathcal{LP}$ (see Figure 4.1), and by standard combinatorics, one may obtain the corresponding $dec_{\mathcal{E}_{Maj}}$ and $rep_{\mathcal{E}_{Maj}}$, and $dec_{\mathcal{LP}}$ and $rep_{\mathcal{LP}}$, respectively. Figure 4.2 presents such values, for different numbers of values proposed

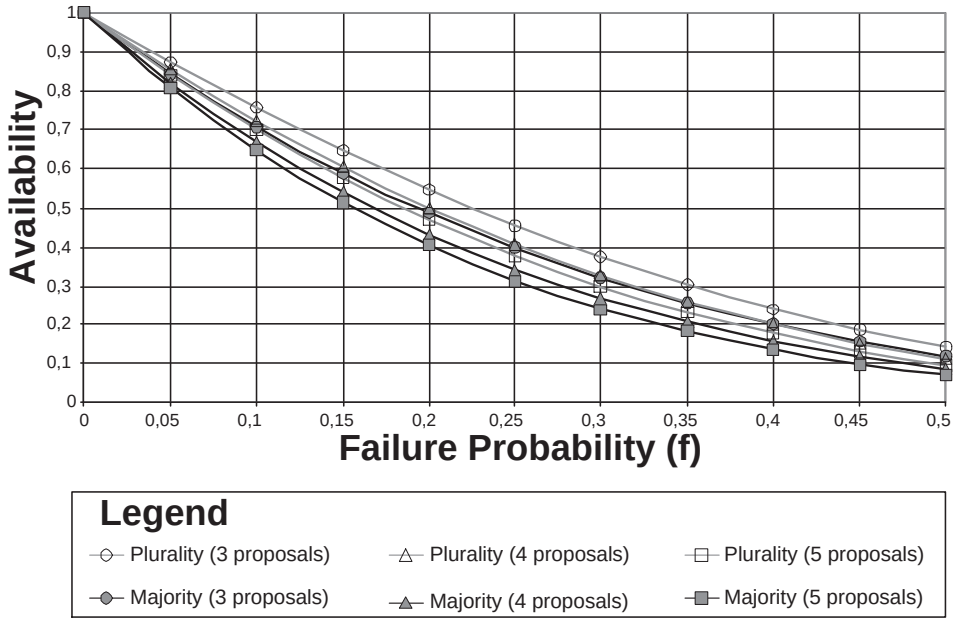


Figure 4.3: Availability of $\mathcal{LP}$ and $\mathcal{E}_{Maj}$, in a system with five sites (3 to 5 concurrent values proposed). $\mathcal{LP}$ yields higher availability than $\mathcal{E}_{Maj}$ for any failure probability and any number of proposals.

($z = |Val|$). As expected, $rep_{\mathcal{LP}}$ is always zero, as $\mathcal{LP}$ always decides in the first election, provided enough votes are collected. Further, $dec_{\mathcal{E}_{Maj}}$ and $dec_{\mathcal{LP}}$ are similar when z is either one or two; in this case, the q-vote configurations of $\mathcal{LP}$ with more than one anti-quorum are of no use, and thus, in practice, $\mathcal{LP}$ induces a similar behavior as $\mathcal{E}_{Maj}$.[6]

When $z \geq 3$, $dec_{\mathcal{LP}}$ is higher than $dec_{\mathcal{E}_{Maj}}$, while $rep_{\mathcal{LP}}$ is lower than $rep_{\mathcal{E}_{Maj}}$. Intuitively, this suggests the following trade-off. On the one hand, the higher $dec_{\mathcal{LP}}$ means that *A* will more easily satisfy *decide-condition(w)*, (i.e. with less votes), thus increasing availability and performance.

On the other hand, when one considers *repeat-condition*, the effect of larger ECs is inverse. The higher $dec_{\mathcal{E}_{Maj}}$ allows *A* to more easily fulfill *repeat-condition*. A *repeat* outcome is useful in situations where the local vote set has reached a state where *decide-condition(w)* depends on the votes of a number of inaccessible sites (e.g. permanently faulty sites). In such a situation, a *repeat* outcome triggers a new election, which acts as a new opportunity for decision; in spite of the inaccessible sites, the new election may eventually lead to a different local vote configuration that finally fulfills *repeat-condition*. If, instead, *repeat-condition* is not met, the election outcome is halted until new votes arrive, which may never happen. There-

[6] This is easy to understand from Figure 4.1, since the only q-vote configurations that differentiate $\mathcal{LP}$ from $\mathcal{E}_{Maj}$ have three or more vote sets (one quorum plus two to three anti-quorums); hence, they only become relevant for $z \geq 3$.

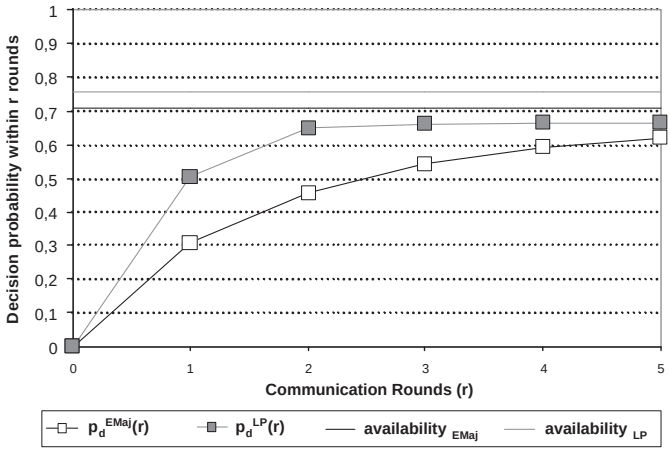


Figure 4.4: Decision probabilities within r rounds of $\mathcal{LP}$ and $\mathcal{E}_{Maj}$ ECs, assuming $p_f = 0.1$, $h = 0.1$, and 3 concurrent values proposed ($z = 3$).

fore, the higher $rep_{\mathcal{E}_{Maj}}$ increases availability, at the cost of more elections (hence, more communication rounds).

We can now analytically deduce the effective outcome of the trade-off on availability and performance. For the results that we present next, we assume the $\mathcal{M}_p$ model for network inaccessibility (see Section 4.1.2).

We start by analyzing availability. By calculating the expression for availability (from Theorem 2), using the probability distributions in Figure 4.2 and varying the failure probability, we obtain the results in Figure 4.3. They show that $\mathcal{LP}$ achieves higher availability for all values of p_f, no matter how many values are proposed. Hence, in what concerns availability, the effect of a higher $dec_{\mathcal{LP}}$ always dominates the effect of a higher rep_{MajEp}.

We now turn our attention to performance, by analyzing the probability of decision within a given number of rounds (using the expression from Theorem 3). As an example, we analyze a particular case, where $p_f = 0.1$ and $h = 0.1$ (where h is the probability that, in the $\mathcal{M}_p$ model, the partition to which A belongs to does not include a given correct site). Figures 4.4, 4.5 and 4.6 present the corresponding performance results of $\mathcal{E}_{Maj}$ and $\mathcal{LP}$. We can see that, in this case, the effect of a higher $dec_{\mathcal{LP}}$ always dominates the effect of a higher rep_{MajEp}, as $\mathcal{LP}$ always achieves higher performance. We have obtained similar results for varying failure probabilities and h.

The results presented above support the statement that $\mathcal{LP}$ achieves higher availability and performance than $\mathcal{E}_{Maj}$. Such a statement has guided the next sections, which propose to solve the problem of update commitment using epidemic quorum systems based on $\mathcal{LP}$.

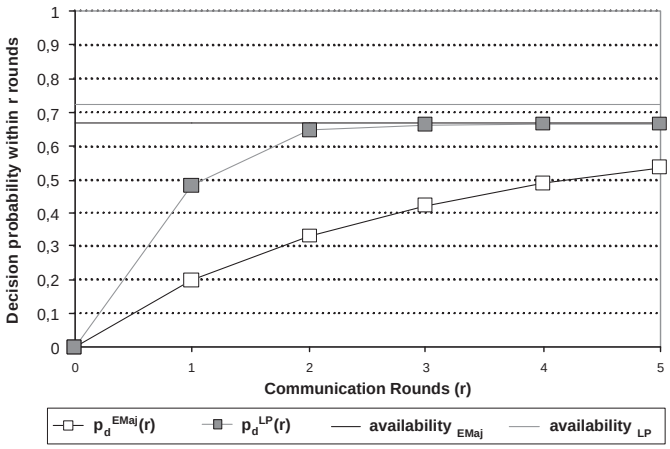


Figure 4.5: Decision probabilities within r rounds of $\mathcal{LP}$ and $\mathcal{E}_{Maj}$ ECs, assuming $p_f = 0.1$, $h = 0.1$, and 4 concurrent values proposed ($z = 4$).

4.1.4 Summary

Epidemic quorum systems are a basic tool for highly available update commitment weakly connected networks, e.g. mobile or sensor networks, where connected quorums are improbable. Although recent work has proposed epidemic quorum algorithms, neither were epidemic quorum systems well defined nor was their liveness well studied.

We formalize epidemic quorum systems and provide a generic characterization of their availability and performance. Our contribution highlights previously undocumented trade-offs that arise in epidemic quorum systems. Further, to the best of our knowledge, it provides the first analytical results that compare proposed epidemic quorum systems. Finally, the formalism serves as a framework for the definition of novel epidemic quorum systems, and for subsequent obtention of new results concerning these systems.

With regard to the present thesis, the obtained results allow us to formally determine which proposed epidemic quorum system is best to base our commitment protocols on. More precisely, the results confirms the intuition that plurality-based systems provide better availability and performance than majority-based ones. We follow this option in the next chapters.

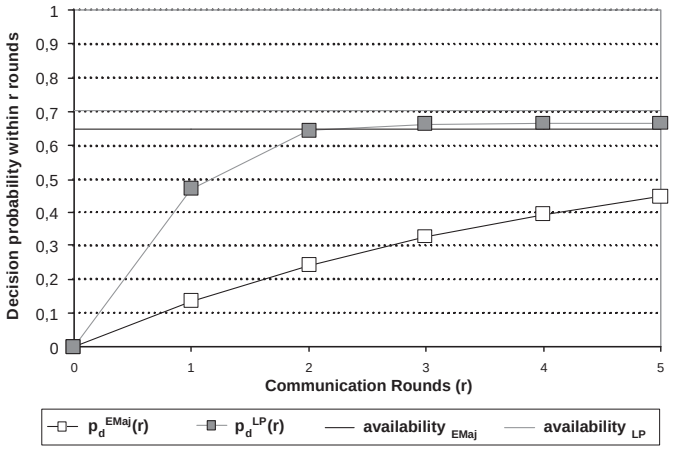


Figure 4.6: Decision probabilities within r rounds of $\mathcal{LP}$ and $\mathcal{E}_{Maj}$ ECs, assuming $p_f = 0.1$, $h = 0.1$, and 5 concurrent values proposed ($z = 5$).

4.2 Version Vector Weighted Voting Protocol

We now apply epidemic quorum systems to solve the problem of commitment in optimistic replication systems in an efficient and fault-tolerant manner. This section proposes Version Vector Weighted Voting Protocol (VVWV), a commitment protocol that is based on a novel extension of the epidemic quorum algorithm that the previous section introduced. VVWV may be used with any epidemic coterie (EC).

We take the results in Section 4.1.3 into account, which show that the linear plurality EC ($\mathcal{LP}$) has higher availability and performance than the majority EC ($\mathcal{E}_{Maj}$). Accordingly, we adopt the former EC in the following presentation of VVWV. Nevertheless, VVWV may easily be changed to adopt any EC. Furthermore, VVWV implements epidemic coteries by using weighted voting. As we explain in the following sections, weighted voting allows for easy self organization of the static set of replicas.

VVWV handles update commitment when explicit semantic information about the updates is not available. In the lack of such information, VVWV exclusively relies on the happens-before relation among updates. VVWV assumes the following reasonable premiss:

> *Any update issued at some replica, upon its tentative value, is semantically compatible with the sequence of updates that produce that very tentative view.*

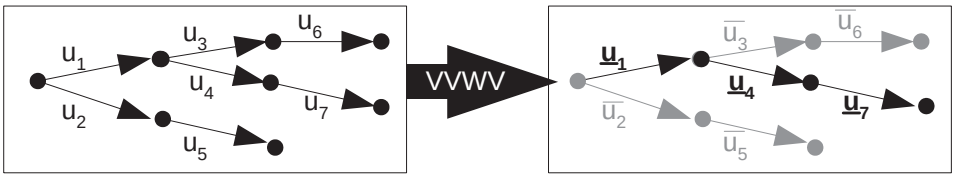


Figure 4.7: Example of semantic-oblivious commitment, the policy adopted by VVWV. From an ever-growing tree of updates, whose edges are the happens-before relations between them, VVWV eventually agrees on a single path of updates linking the root to a leave to be committed. All updates concurrent (by happens-before) with such a path abort.

As a direct consequence, update conflicts may only arise from concurrent updates. Accordingly, VVWV follows the conservative approach of assuming that update concurrency (by happens-before) always results in conflicting update schedules.[7] In the presence of a set of concurrent updates, it guarantees that committed schedules are always sound by always aborting all updates in the set but

[7] Although, in practice, two concurrent updates are not necessarily semantically conflicting.

one. Hence, the distributed system produces a tree of updates, related by happens-before. VVWV eventually commits the updates along a single connected path from the root to one of the tree's leaves, aborting every other update. Figure 4.7 depicts an example.

One possible solution would be to directly apply the epidemic quorum algorithm that Section 4.1.1.3 introduced to iteratively decide which update to commit at each level of the tree. Considering the above example, we would run the epidemic quorum algorithm to elect one update among u_1 and u_2; then, assuming u_1 wins, rerun the algorithm to elect one update among u_3 and u_4 (or, if u_2 wins, to elect u_5); and so forth. This is the approach that the Deno replicated system [Kel99] follows.

In some cases, replicas issue more than one tentative update prior to its commitment decision. This is increasingly probable when connectivity is weak and commitment has to wait for significant delays before the other replicas become accessible. In the meantime, a replica may continue to issue a sequence of happens-before-ordered tentative updates. In this case, the previous approach, which requires one election to commit each update, is clearly not appropriate.

Instead, it is more efficient to commit sequences of multiple updates related by happens-before in a single election. Intuitively, this requires extending the epidemic quorum algorithm so as to support multiple-update votes. More than simply expressing the intention of voting for a given tentative update, a multiple-update vote might also express the fact that, if the first update does become elected, then the voter also supports a second update, and so forth. Using such richer vote information, an adequate extension of the epidemic quorum algorithm would be able to, when possible, elect more than one update in only one election round.

VVWV achieves such a goal by employing version vectors to identify election candidates. The flexibility brought by version vectors allows a sequence of one or more updates to run for the current election as a whole. The candidate is represented by the version vector corresponding to the tentative version obtained if the entire update sequence was applied to the replica. As the next sections explain, the election algorithm of VVWV relies on the expressiveness of version vectors to decide if the update sequence or a prefix of it are to become committed. Consequently, candidates consisting of one or more happens-before-related updates may commit on a single distributed election round. In weakly connected network environments, where such update patterns are expectably dominant, a substantial reduction of the update commitment delay is therefore achievable.

A version vector holds one counter per site that replicates the object being considered. For presentation simplicity, and without lack of generality, we assume only one object. Furthermore, when some site A replicates the object with a replica a, we denote the counter in some version vector vv that corresponds to A as $vv[a]$ (instead of the correct notation, $vv[A]$).

We recall our notation when reasoning about version vectors. Given two version vectors x and y, we write $x \leq y$ to denote that $\forall i, x[i] \leq y[i]$, and $x < y$ to denote that $x \leq y$ and $\exists j : x[j] < y[j]$). Further, $x \parallel y$ means that x and y are not

comparable, that is, neither $x \leq y$ nor $y \leq x$.

The remainder of the section is as follows. Section 4.2.1 addresses the use of a weighted voting approach based on a plurality coterie. Section 4.2.2 describes the state maintained at each replica and the two distinctly consistent views that the protocol offers of each replica. Section 4.2.3 addresses the storage of tentative updates and their corresponding commitment upon a replica value. Section 4.2.4 describes the epidemic flow of consistency information and Section 4.2.5 defines how candidates are elected. Finally, Section 4.2.7 summarizes.

4.2.1 Epidemic Weighted Voting with Plurality Coteries

VVWV employs the approach of *weighted voting with fixed per-object currency* [Kel99] to define epidemic coteries. Each replica a is associated with a given currency, $currency_a$, which determines its weight in the elections. This approach allows easy self organization of the system, as weight needs to be redistributed as network properties change [CK02].

Weights may be easily redistributed by atomic pair-wise interactions, where replicas exchange fractions of their currency. Replicas need not have a global knowledge of the currency that each other replica owns. Instead, the may simply rely on the property that the sum of the currencies held by all replicas remains equal to (or lower than) a fixed amount (e.g. 1), in order to take safe commitment decisions. Hence, some replica may deliver some amount of its current currency to some other replica by by simply exchanging the latter atomically; no coordination with the remaining replicas is necessary.

We define the epidemic coterie that VVWV uses in terms of the currencies held by replicas. We denote such a coterie as $\mathcal{WLP}$. It varies dynamically as currencies are exchanged among replicas.

$\mathcal{WLP}$ is based on a variant of the Linear Plurality EC ($\mathcal{LP}$, which we defined in Section 4.1.1.2), and was initially proposed by Keleher in [Kel99]. When currencies are evenly divided among replicas, the epidemic coterie used by VVWV is equivalent to the Linear Plurality EC ($\mathcal{LP}$) defined in Section 4.1.1.2. The analytical results we obtain in Section 4.1 justify such a choice over other epidemic coteries proposed in literature.

Prior to defining $\mathcal{WLP}$, we need to define the auxiliary notion of set currency:

Definition 11. *Set Currency.*

Let s be a set of replicas. The set currency of s, denoted $cur(s)$, is given by $cur(s) = \sum_{a \in s} currency_a$.

We definition the $\mathcal{WLP}$ EC as follows:

Definition 12. *Weighted Linear Plurality EC.*

The Weighted Linear Plurality EC, $\mathcal{WLP}$, is defined by the set of q-vote configurations, such that, for each q-vote configuration, $qc = \langle Q^{qc}, \{A_1^{qc}, A_2^{qc}, .., A_n^{qc}\}\rangle \in \mathcal{WLP}$:

1. $\forall k, cur(Q^{qc}) > cur([\emptyset]_{qc})$ *or* $(cur(Q^{qc}) = cur([\emptyset]_{qc}) \wedge Q^{qc} <_{break} [\emptyset]_{qc}$ *and*
2. $\forall k, cur(Q^{qc}) > cur([A_k^{qc}]_{qc})$ *or* $(cur(Q^{qc}) = cur([A_k^{qc}]_{qc}) \wedge Q^{qc} <_{break} [A_k^{qc}]_{qc}$, *and*
3. $\forall qd \in \mathcal{WLP}, qc \not\succ qd$.

The above $<_{break}$ relation among vote sets is the same as defined in Section 4.1.1.2.

Having defined the epidemic coterie, it is straightforward to use the algorithm described in Section 4.1.1.3 to devise an update commitment protocol. In such a protocol, each tentative update that a replica issued would be submitted as a candidate to the current epidemic election.[8]

However, such a protocol would not efficiently handle sequences of multiple updates related by happens-before. In such a case, it would require one election per update, where intuitively it would seem possible to elect a whole sequence in a single election round. The next sections show how VVWV extends the above algorithm so as to attain such a goal.

4.2.2 Replica State and Access to Committed and Tentative Views

Each replica a maintains the following state:

- $decidedForCommit_r$, which consists of a version vector that identifies the most recent decided-for-commitment version that a is currently aware of;
- $votes_a[1..N]$, which stores, for each replica $k = 1, 2, .., N$, the version vector corresponding to the candidate voted for by k, as known by a; or $\perp$, if the vote of such a replica has not yet been known to a;
- $cur_a[1..N]$, which stores, for each replica $k = 1, 2, .., N$ whose vote replica a has knowledge of, the currency associated with such a vote;
- $committed_a[1..c_a]$, which stores each committed update at a according to the agreed commitment order (where c_a denotes the number of committed updates at a);

Each site offers two possibly distinct views over the value of a replica a: the committed view and one tentative view. The first reflects a strongly consistent value of the replicated object that is obtained by the ordered application of the updates in $committed_a$. On the other hand, the tentative view exposes a weakly consistent value that corresponds to the candidate version that is currently voted by the local replica, $votes_a[r]$.

[8] Where *tentative updates* would correspond to what the notation in Section 4.1 calls *proposed values*.

Write requests from applications are performed upon the tentative view. Each write request results in a new tentative update, u, that is appended to the end of the r-schedule as executed (i.e. as $\underline{u}$). The resulting new r-schedule causes the replica to propose and vote for a new candidate. More precisely, the *proposeSch* (see Section 3.2.5) is, in VVWV, specified as follows:

(adv_a advances the counter corresponding to a in the supplied version vector by one.)

Algorithm 11 $proposeSch(schedules)$ at replica a, in VVWV

1: **if** $votes_a[r] = \perp$ **then**
2: $\quad votes_a[r] \leftarrow adv_a(decidedForCommit_r)$.
3: $\quad cur_a[r] = currency_a$.
4: **else**
5: $\quad votes_a[r] \leftarrow adv_a(votes_a[r])$

In case a has not voted yet, then it casts a vote for the candidate representing the new tentative schedule, which includes the new update, u, which was issued upon the current committed version. Otherwise, it means that u is issued upon the value resulting from the ordered application of the tentative updates of the current vote of a. In this case, its vote is extended to a candidate that represents u and the tentative updates that precede it.

4.2.3 Update Propagation and Commitment

The protocol proposed hereafter requires only that the replicated system ensures the prefix property. Apart from that requirement, VVWV is orthogonal to the the issues of actual propagation and storage of tentative updates.

VVWV does not impose the decision of whether to transfer and store, at each individual replica, (1) the tentative updates belonging to every candidate in the current election or, alternatively, (2) only those concerning the replica's own candidate. This means that, at the time a replica determines that a given candidate has won the election and, thus, its updates should be committed, such updates may not be immediately available (if the replica adopts mode (2)). Instead, they will be eventually collected through later anti-entropy sessions with other replicas.

Consequently, there may occur a discrepancy between the most recent version that VVWV has decided at a given replica a and the actual stable value that is locally accessible at a. In fact, the number of updates in $committed_a$, noted c_a, may be lower than the number of updates that have actually been determined by VVWV as belonging to the decided-for-commitment path. In such a case, the replica's committed value does not yet reflect the most recent decided version a is aware of.

As a consequence, VVWV is flexible enough to support sites with differing memory limitations. On one hand, sites with rich memory resources may store every update associated with each candidate, hence being able to immediately gain

access to the most recent known decided value as each new stable version is determined by VVWV. On the other hand, memory-constrained devices may opt to restrict themselves to storing only the updates of their own candidate and, thus, allow for occasional delays in the availability of the most recent decided value when rival candidates win an election. In either case, however, the efficiency of VVWV in taking commitment decisions is not affected. Both strategies may transparently co-exist in a system of replicas of the same logical object.

From the viewpoint of VVWV, the procedure for committing a sequence of updates $u_1,..,u_m$ is therefore comprised of the following steps:

1. For each update, $u_k(k = 1..m)$, $committed_a[c_a + k] \leftarrow u_k$;
2. $c_a \leftarrow c_a + m$;

The above procedure occurs once $u_1,..,u_m$ become both available at replica a and each $\underline{u}$ has been decided for commitment by VVWV.

4.2.4 Anti-entropy

Voting information is propagated through the system via anti-entropy sessions established between pairs of accessible sites. Voting information may be piggybacked to the anti-entropy sessions that Section 3.2.4 described. During an anti-entropy session, a requesting site, holding replica A, updates its local election knowledge with information obtained from another site, holding replica B. In case B has more up-to-date election information, it transfers such information to A. Furthermore, if A has not yet voted for a candidate that competes against the one voted for by B, A accepts the latter, thus contributing to its election.

At each anti-entropy session, the following procedure is carried out atomically:

1. If $decidedForCommit_a < decidedForCommit_b$ then
   (a) $decidedForCommit_a \leftarrow decidedForCommit_b$;
   (b) Let $toCommit_a = u_1,..,u_m$ be the totally ordered sequence of uncommitted updates that, starting from the currently committed value at a, produces the new $decidedForCommit_a$ version. If a prefix of $toCommit_a$, $u_1,..,u_k(k \leq m)$ is locally available at a, then commit such updates in that order;
   (c) If $c_a < c_b$ then commit update sequence $committed_b[c_a + 1],..,committed_b[c_b]$.
   (d) $\forall k$ s.t. $votes_a[k] || decidedForCommit_a$ or $votes_a[k] \leq decidedForCommit_a$, then $votes_a[k] \leftarrow \perp$;
2. If ($votes_a[a] = \perp$ and $decidedForCommit_a < votes_b[b]$) or $votes_a[a] < votes_b[b]$ then
   $votes_a[a] \leftarrow votes_b[b]$ and $cur_a[a] \leftarrow currency_a$;

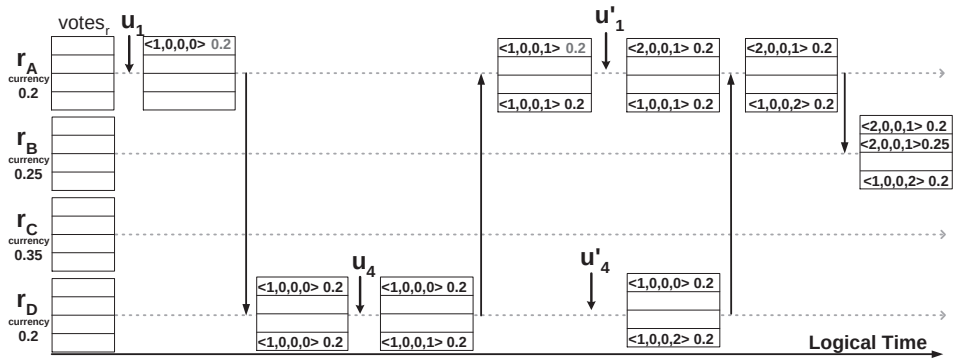


Figure 4.8: Example of update generation and anti-entropy: four replicas with unevenly distributed currencies start from a common initial stable version $decidedForCommit_r = \langle 0,0,0,0 \rangle$.

3. $\forall k \neq a$ s.t. ($votes_a[k] = \perp$ and $decidedForCommit_a < votes_b[k]$) or $votes_a[k] < votes_b[k]$ then

   $votes_a[k] \leftarrow votes_b[k]$ and $cur_a[k] \leftarrow cur_b[k]$.

The first step ensures that, in case *b* knows about a more recent decided-for-commitment version, *a* will adopt it. This means that *a* will regard the elections that originated the newly decided-for-commitment version as completed (1a) and, therefore, commit the winner updates that are available at the moment.

In (1b), any winner updates that are already locally stored (tentatively) are committed. After that, the set of committed updates held by *b* that have not yet been committed at replica *A* are collected and committed by the latter (1c). Finally, further elections are prepared by keeping only the voting information that will still be meaningful for their outcome (1d). Namely, these are the votes on candidates that succeed (by happens-before) the decided-for-commitment version.

As a second step of anti-entropy, *a* is persuaded to vote for the same candidate as the one voted by *b*, provided that *a* has not yet voted for a competing candidate (2). Subsequently, *a* updates its current knowledge of the current election with relevant voting information that may be held by *b* (3). Namely, *a* stores each vote that it is not yet aware of or whose candidate is more complete than the one it currently has knowledge of.

Figure 4.8 depicts an example with four replicas of a given object. After some update activity, a conflict occurs between u'_1 and u'_4; hence, two rival candidates end up running for the election.

4.2.5 Election Decision

The candidates being voted in an election represent update paths that traverse through one of more versions beyond the initial point defined by the decided-for-commitment version, $decidedForCommit_r$. These possibly divergent candidate up-

date paths may share common prefix sub-paths. The following definition expresses this concept.

Definition 13. *Maximum Common Version.*

Given two version vectors, v_1 and v_2, their maximum common version is a version vector, $mcv(v_1,v_2)$, s.t. $\forall k, mcv(v_1,v_2)[k] = min(v_1[k],v_2[k])$.

For simplicity, we represent $mcv(mcv(mcv(v_1,v_2)),...),v_m)$ by $mcv(v_1,v_2,..,v_m)$.

Theorem 4. *Let $v_1,..,v_m \in votes_a$, be one or more candidate versions known by replica a, each denoting a tentative update path starting from the decided-for-commitment version, $decidedForCommit_r$. Their maximum common version, $mcv(v_1,..,v_m)$, constitutes the farthest version of an update sub-path that is mutually traversed by the update paths of $v_1,..,v_m$. Complementarily, the total currency voted for such a common sub-path is obtained by $voted_a(mcv(v_1,..,v_m)) = cur_a[v_1] + ... + cur_a[v_m]$.*

Proof. Assume, by contradiction, that $m = mcv(v_1,..,v_m)$, is not the farthest version of an update sub-path that is shared by the update paths of $v_1,..,v_m$; instead, such a version is given by $m' \neq m$. By definition of *mcv*, $m' \not> m$; otherwise, $m' \leq v_i, \forall i$, wouldn't be verified. So, $m' < m$, which implies that $\exists k$ s.t. $m'[k] < m[k] \leq v_1[k]$ and $m'[k] < m[k] \leq v_2[k]$. Since m' identifies the farthest common sub-path, the differences between $m'[k]$ and $v_1[k]$, as well as $m'[k]$ and $v_1[k]$, must have respectively resulted from concurrent tentative updates generated by replica k. However, replica k is not allowed to issue concurrent updates when holding the same $decidedForCommit_k$: if $votes_k[k] = \perp$ (Section 4.2.2), it means that $\nexists i \neq k$ s.t. $votes_k[i]$ represents any tentative update from k. By anti-entropy, this is also verified at any other replica. □

VVWV is responsible for progressively determining common sub-paths of candidate versions that manage to obtain a plurality of votes. Before describing this decision, we define the notion of currency collectable by a candidate c:

$$collectable_a(c) = \sum(cur[k] : votes_a[k] = \perp \vee votes_a[k] < c).$$

The currency collectable by a candidate c is obtained from the currencies of the votes that, despite not supporting c, may evolve to such a condition; namely, votes on candidates that happen-before c or $\perp$.

We may now define the election condition of VVWV.

Definition 14. *Let w be a version vector s.t. $w = mcv(w_1,..,w_m)$ where $w_1,..,w_m \in votes_a$ and $1 \leq m \leq N$. w wins an election when:*

1. *$voted_a(w) > 0.5$, or*
2. *$\forall l$ s.t. $l = mvc(l_1,..,l_k), l_1,..,l_k \in votes_a$, $1 \leq k \leq N$, and $l \parallel w$, and also $\forall l$ s.t. $l = mcv(l_i : l_i \in votes_a \wedge l_i < w)$,*

Figure 4.9: Election decision for replica r_B at the final state of the example in Figure 4.8. Candidate $\langle 1,0,0,1\rangle$ has collected a plurality of votes and, thus, u_1 and u_4 will be committed in that order.

(a) $voted_a(w) > voted_a(l) + collectable_a(l)$, *or*

(b) $voted_a(w) = voted_a(l) + collectable_a(l)$ *and* $w \prec_a l$.

The above rules state the conditions that guarantee that a candidate has collected sufficient votes to win an election. Essentially, the rules imply that each individual update that becomes elected acquires the required quorum of votes (and its concurrent updates the necessary anti-quorum of votes) of some q-vote configuration from the $\mathcal{WLP}$ epidemic coterie.

The votes may constitute a majority, when the amount of currency voted on the winning candidate surpasses 0.5; or a plurality, when the voted currency is greater than the maximum potentially obtainable currency of any other rival candidate. This includes not only common sub-paths of *known* concurrent candidates; but also a hypothetical *unknown* concurrent candidate for which non-concurrent votes can still potentially evolve to.

It should be noted that only the non-concurrent votes v_i such that $v_i < w$ may evolve to $v_i' > v_i$ where $v_i' \parallel w$. Instead, this is not the case for votes v_j such that $v_j \geq w$, as w will happens-before any $v_i' > v_j$.

Finally, the case of ties is decided by the $\prec_a$ relation; we discuss it shortly, in Section 4.2.6.1.

Determining if a candidate has won an election depends exclusively on information that is locally available at each replica. This means that, once having collected enough voting information, a given replica is able to decide, by its own, to commit a candidate version that locally fulfills the election winning conditions. Hence, update commitment is accomplished in a purely decentralized manner. Figure 4.9 illustrates the different maximum common versions that replica r_B analyzes after the example in Figure 4.8 in order to determine the election outcome; r_B is able to decide that updates u_1 and u_4 should be committed as soon as available at r_2.

After finding a new winner version vector, w, a replica a atomically takes the following steps to accept the election decision and prepare for the next election:

1. $decidedForCommit_r \leftarrow w$;
2. $\forall v_k \in votes_a$ s.t. $v_k \| w$ or $v_k \leq w$, $votes_a[k] \leftarrow \perp$;

3. Let $toCommit_a = u_1, .., u_m$ be the totally ordered sequence of uncommitted updates that, starting from the currently committed value at a, produces the new $decidedForCommit_r$ version. Function *getNewDecidedSch* (see Section 3.2.5 returns the prefix of $toCommit_a$, $u_1, .., u_k (k \leq m)$ that is locally available at a.

After accepting the election result by setting the winning version as the new decided-for-commitment version, the second step resets all the defeated candidates to $\perp$. Depending on the local availability of the updates that belong to the winning candidate, they may be returned in a new stable schedule by *getNewDecidedSch* and, consequently, then commit into the replica's stable value (as defined in the generic protocol in Section 3.2.5); otherwise, further anti-entropy sessions will ensure that such updates are eventually collected and, then, committed. A new election can then take place.

4.2.6 Correctness

VVWV satisfies the criterion of eventual consistency (EEC). Namely:

1. for every replica a, sch_a is sound; and
2. at any moment, for each replica a, there is a prefix of sch_a, denoted $allStable_r$, that is semantically equivalent to a prefix of sch_b, for every other replica b; and
3. $allStable_r$ grows monotonically by suffixing over time; and
4. $allStable_r$ is sound; and
5. for every update u, generated at any replica, and for every replica a, after a sufficient finite number of anti-entropy sessions, $allStable_r$ either includes $\underline{u}$ or $\overline{u}$.

We do not present an extensive proof of the above statement. VVWV clearly ensures the first and third conditions of EEC. It is easy to prove the second and fourth conditions from the following lemma.

Lemma 3. *After all elections have been completed at every replica and all updates belonging to the resulting decided-for-commitment path have been committed at every replica, $\forall r, t$, replica a has committed the same ordered sequence of updates as t.*

Proof. We present an outline of the proof, which is based on the proof for the Deno replicated system [Kel99]. Assume that all replicas start with a common decided-for-commitment value, $decidedForCommit_0$. It follows directly from the protocol that, if $votes_a[j] = k$ for any r, j, k, then for any l such that $decidedForCommit_l = decidedForCommit_r = decidedForCommit_0$, $votes_l[j]$ will either be $\perp$

or v, where v is comparable with k (i.e., $v \leq k$ or $v > k$). Let $S = v_1, .., v_m$, where $v_1, .., v_m \in votes_a$, and $w = mcv(v_1, .., v_m)$. Assume now that a decides that w wins the election, thus setting $decidedForCommit_r \leftarrow w$; hence, the currency collected by w prevented any rival $c \parallel w$ to be declared winner. For each replica m such that $decidedForCommit_0 \leq decidedForCommit_m < w$, it can be shown that, for each $s \in S$, $votes_m[s]$ must either be $\perp$ or comparable with w; this prevents any rival $c \parallel w$ to be decided winner at m. By anti-entropy, m will eventually receive enough voting information to determine that w has collected a plurality, or directly receive the outcome of the election from another replica p having $decidedForCommit_p \geq w$. As the totally ordered set of updates that produce w is eventually propagated to every replica, they will be accordingly be committed in that order, the same at every replica. □

4.2.6.1 Tie Breaking with Multiple-Update Candidates

The $\mathcal{WLP}$ EC, which we define in Section 4.2.1, relies on an order relation among vote sets (the $<_{break}$ relation) in order to resolve situations where two vote sets obtain the same currency; i.e. to break tied elections. In Section 4.2.1, we have proven that the $<_{break}$ relation ensures that $\mathcal{WLP}$ is an EC. This implies that, in the presence of a tie, $<_{break}$ will always break it in favor of the same candidate; no matter which replica (whose partial view of the current vote configuration of the system may differ with other replicas) we consider.

For the moment, we only know that the above proposition is true when we use the basic epidemic quorum algorithm, which Section 4.1.1.3 defines, and which can only handle single-update (or single-value) candidates. Nevertheless, VVWV reasons about multiple-update candidates. It is, then, necessary to devise an order relation, which we denote $\prec_a$, among vote sets on multiple-update candidates that still ensures that ties are broken consistently at every replica.

In this section, we define $\prec_a$ and prove its safety

A tie between candidates x and y is decided at replica a by choosing x if the partial relation $x \prec_a y$ holds. Intuitively, $x \prec_a y$ if a replica v_x contributes with its vote to the candidate x and no other voter with a lower identifier than v_x contributes, or may ever contribute while x is still a candidate, with its vote to candidate y. Formally, $\prec_a$ is defined as follows:

Definition 15. *Tie breaking relation. Let x and y be two version vectors so that $x \parallel y$ and that, at replica a, $\exists\, i, j : decidedForCommit_r < x \leq votes_a[i]$ and $decidedForCommit_r < y \leq votes_a[j]$.*

We say that a tie between x and y at a is broken in favor of x, or $x \prec_a y$, if and only if $\exists v_x$ such that

1. *$votes_a[v_x] > mcv(x,y)$ and $votes_a[v_x] \geq x$, and*
2. *$\forall v_y < v_x$, $votes_a[v_y] > mcv(x,y)$ and $votes_a[v_y] \parallel y$.*

Since $\prec_a$ is a partial relation, it may happen that, at some point in time, two candidates x and y are not related by $\prec_a$. In this case, the tie breaking decision is deferred until sufficient voting information is received to decide either $x \prec_a y$ or $y \prec_a x$ (or until the tie ceases to be verified). The following theorem ensures the correctness of VVWV when election decisions are made by breaking ties using $\prec_a$. Intuitively, it states that, if a tie between two rival candidates x and y at a given election is broken at a replica a by deciding the victory of x, then no different tie breaking decisions will ever be obtained at any replica.

Theorem 5. *If, at some moment, $x \prec_a y$ is verified at a replica a, then $y \prec_i x$ will never be verified at any replica $i = 1..N$.*

Proof. Assume that, at a given moment, $x \prec_a y$ and that v_x is defined as in Definition 3. Then, it follows directly from the definition of $\prec_a$ that $y \not\prec_a x$. Further, assume, by absurd, that $y \prec_{r'} x$, where r' denotes a in some point in the future. This means that either (a) $votes_{r'}[v_x] \not\geq x$, or (b) $\exists v_y < v_x : votes_{r'}[v_y] > mcv(x,y)$ and $votes_{r'}[v_y]$ is comparable to y, i.e. $votes_{r'}[v_y] < y$ or $votes_{r'}[v_y] \geq y$ (where previously, by definition of $\prec_a$, $votes_a[v_y] \parallel y$ since $x \prec_a y$). According to the protocol, $\forall i, votes_a[i] \neq \perp$, $votes_a[i]$ may only be changed, at each single step, to $v > votes_a[i]$, or to $\perp$; it follows from this that (a) and (b) imply that, at some step, $votes_a[v_x] \leftarrow \perp$, or $votes_a[v_y] \leftarrow \perp$, respectively. Such an assignment is caused by a new $decidedForCommit'_r$ version at a such that $decidedForCommit'_r \not\leqslant mcv(x,y)$ (otherwise, neither $votes_a[v_x] > mcv(x,y)$ nor $votes_a[v_y] > mcv(x,y)$ would have been changed). So, one of two cases must have occurred: (1) $decidedForCommit'_r \parallel mcv(x,y)$, and hence $decidedForCommit'_r \parallel x$ and $decidedForCommit'_r \parallel y$; or (2) $decidedForCommit'_r > mcv(x,y)$, and thus (by a corollary of Theorem 1, and as $x \parallel y$) either $decidedForCommit'_r \parallel x$ or $decidedForCommit'_r \parallel y$ (or both). It can easily be proven that VVWV ensures that $\forall\ i, votes_a[i] > decidedForCommit'_r$; so, neither both x and y will be candidates with $decidedForCommit'_r$, as at least $x \parallel decidedForCommit'_r$ or $y \parallel decidedForCommit'_r$, and therefore $y \not\prec_a x$; this contradicts the initial hypothesis. A similar reasoning extends the proof to the case of other replicas. □

4.2.7 Summary

We propose a protocol, VVWV, based on epidemic weighted voting for achieving such a goal with better availability than traditional primary commit approaches. We improve previous epidemic weighted voting solutions by allowing commitment of multiple, happens-before-related updates at a single distributed election round.

4.3 Decentralized Commitment With Rich Semantics

VVWV does not need any explicit semantic information about the updates as it relies on the happens-before relation among updates. VVWV follows a conservative approach, which assumes that any concurrent updates (by happens-before) may conflict. Hence, in the presence of a set of concurrent updates, it guarantees that committed schedules are sound by always aborting all updates in the set but one. However, such a conservative approach has the drawback of aborting updates unnecessarily.

In some cases, some richer semantic knowledge about the issued updates is available. For instance, Bayou's applications provide dependency checks [TTP+95] along with each update. A dependency check is an application-specific conflict detection method, which receives the value resulting from a given schedule as input. The output tells whether the update may be safely appended to the end of such a schedule or not, which respectively yields compatibility or conflict relations between the update and the schedule. Even richer semantic information is available, in IceCube [KRSD01] or the Actions-Constraints Framework (ACF) [SBK04]. The approach of IceCube and the ACF reifies semantic relations as a graph where nodes correspond to updates, and edges consist of semantic constraints of different kinds.

When rich semantic knowledge is available, one would expect the commitment protocol to take advantage of it. The advantage should be twofold. On one hand, richer semantics should minimize aborts needed for combining divergent tentative updates into a sound schedule, to later become decided. On the other hand, richer semantics should accelerate commitment by permitting a weaker-than-total order among the stable schedules of the system. We illustrate both potential improvements with an example.

Consider the classic example of a bank account whose state is replicated optimistically. The bank account is co-owned by Alice and Bob, each requesting account operations (i.e., writes) to a distinct replica. The bank account is modified by debit and credit operations, and its semantics require that it must never have a negative balance. Assume that, right before a partition disconnects both replicas, they have an identical value, with a balance of 100 euro. During disconnection, Alice requests to debit the account by 100 euro, which results in tentative update u_1. Concurrently, Bob requests the following operations, in this order: to debit the account by 50 euro (u_2), to credit it by 100 euro (u_3), and to credit it by 70 euro (u_4).

If we use VVWV, it eventually decides to either commit u_1 and abort u_2, u_3 and u_4, or vice-versa. However, if richer semantics were available, a commitment protocol could take better decisions. For instance, it could ignore concurrency and order u_1 after u_3, e.g., committing $u_2;u_3;u_1;u_4$, which is sound according to semantics; therefore preventing any aborts, while satisfying the semantic condition of non-negative balances.

Furthermore, the commitment protocol could be more flexible, deciding commitment orders that are weaker than a total order. Recalling our example, the pro-

tocol may notice that, having $u_2;u_3$ committed, no matter the order by which each individual site executes u_1 and u_4, the same correct balance is eventually achieved system-wide. Hence, the commitment protocol may simply agree on a partial order (since u_2 and u_3 need not be ordered), rather than a total order; expectedly, such a weaker agreement should be easier, and thus more rapidly attainable.

Previous work on commitment for semantic optimistic replication, such as Bayou [DPS+94] or IceCube [KRSD01], centralizes agreement at a central site. This is clearly not fault-tolerant, which violates the requirements of our work. Other work decentralizes commitment (e.g., [Kel99]) but ignores semantics, hence ignoring the above potential advantages.

It is difficult to reconcile semantics and decentralization. One possible approach would be to incrementally compute a total order among updates as they are issued, using some decentralized algorithm (e.g. Paxos [Lam98]); then abort any updates that semantically conflict with the preceding schedule in the total order. However this approach still aborts updates unnecessarily, since the generation of the total order is oblivious of semantics. To illustrate, recall the above example and assume the agreed total order is $u_1;u_2;u_3;u_4$; according to semantics, u_2 must abort. Furthermore, such an approach always computes a total order, neglecting opportunities where semantics allow weaker, hence less costly in number of messages, orders.

Naturally, a better approach is to partially order updates taking semantics into account. However, to the best of our knowledge, no previously proposed solution achieves such a goal in a decentralized fashion.

In joint work with Pierre Sutra and Marc Shapiro [SBS07], we have extended VVWV and, assuming that rich semantic knowledge is available, so as to achieve the goal of decentralized semantic commitment. This section describes and discusses such work, which complements our contributions in the previous sections of the chapter.

We consider one already existing formalism for expressing semantics, the Actions-Constraints Framework (ACF) [SBK04], which is able to express a rich semantic repertoire. The result is a protocol that is able to produce the ideal commitment decisions that we illustrate in the initial example above.

Similarly to VVWV, its semantic extension is decentralized, achieving consistency by means of the same epidemic voting approach as VVWV. Instead of agreeing on happens-before-related update sequences, the protocol handles semantic graphs, called *multilogs*, as candidates for commitment. Commitment evolves by incrementally agreeing on candidate sub-graphs as decided. Consequently, and as the next sections show, agreement requires a more complex analysis and comparison of candidates than in the semantic-oblivious case.

We hereafter present an abstract protocol, whose operation depends on the definition of *eligible candidates*, which we leave open. We describe the properties that eligible candidates must satisfy and, as an exercise, propose two possible definitions. As we discuss, the proposed definitions are not entirely satisfactory, either because they do not fully exploit semantics, or because they depend on a single

point of failure. Nevertheless, we advocate that such an analysis lays the ground for an appropriate definition of eligible candidate and, consequently, for solving the decentralized semantic commitment problem.

We should note that the issue of adequate semantic formalisms is out of the scope of the present thesis. In particular, this section does not aim at claiming that the ACF is the most appropriate semantic formalism for the environments that the thesis considers. In the scope of the thesis, the choice of the ACF is merely justified by its very rich semantic expressiveness [SBK04] and by its relevance as a semantic formalism among other existing alternatives (e.g. [KRSD01], [Lam05] or [FG00]).

The remainder of the section is organized as follows. Section 4.3.1 starts by introducing the Actions-Constraints Framework, mapping it against the system model and terminology that we present in Chapter 3. Section 4.3.2 then describes how to generically support optimistic replication with the ACF. The issue of update commitment is left for the subsequent sections. Section 4.3.3 discusses centralized approaches to achieve update commitment in a system relying on the ACF. Sections 4.3.4 and 4.3.5 then addresses the problem of decentralized semantic update commitment, proposing two possible solutions and analyzing their limitations. Finally, Section 4.3.6 summarizes.

4.3.1 Actions-Constraints Framework

The ACF designates a request to execute some logical operation as an *action*. An action corresponds to an update in the terminology introduced in Section 3.2; in the remaining sections, we will use the term action, for coherence with the ACF.

The ACF designates applications and users running at a given site as the local *client*. Clients propose actions, denoted $\alpha, \beta, \ldots \in Actions$. An action might request, for instance, "Debit 100 euros from bank account number 12345." The ACF assumes actions to be unique and distinguishable from one another. Furthermore, the ACF does not differentiate individual objects; hence, we can regard actions as applied to a single universal object that all sites fully replicate.

The central data structure of the ACF is the *multilog*. A multilog is a quadruple $M = (K, \rightarrow, \lhd, \nparallel)$, representing a graph where the vertices K are actions, and $\rightarrow$, $\lhd$ and $\nparallel$ (pronounced NotAfter, Enables and NonCommuting respectively) are three sets of edges called *constraints*.[9] We will explain their semantics shortly.

The current knowledge of site A at time t is the distinguished *site-multilog* $M_A(t)$. Initially, $M_A(0) = (\{\text{INIT}\}, \varnothing, \varnothing, \varnothing)$, and it grows over time, as we will explain later.

In the ACF, a *schedule* S is a sequence of distinct actions ordered by $<_S$ executed from the common initial state INIT. The NotAfter and Enables constraints of a multilog determine the universe of sound schedules that may result from the

[9] Multilog union, inclusion, difference, etc., are defined as component-wise union, inclusion, difference, etc., respectively. For instance if $M = (K, \rightarrow, \lhd, \nparallel)$ and $M' = (K', \rightarrow', \lhd', \nparallel')$ their union is $M \cup M' = (K \cup K', \rightarrow \cup \rightarrow', \lhd \cup \lhd', \nparallel \cup \nparallel')$.

multilog. On one hand, $\alpha \rightarrow \beta$ implies that, if both α and β appear in a given schedule, then the schedule orders β after α. On the other hand, $\alpha \lhd \beta$ means that, if β appears in a given schedule, also does α. NotAfter is, hence, a (non-transitive) ordering relation, while Enables is right-to-left (non-transitive) implication.[10] Antagonism[11] between two actions α and β, i.e., either α or β (or both) must abort, is represented by having: α NotAfter β and β NotAfter α.

The ACF denotes the set of schedules S that are *sound* with respect to multilog M as $\Sigma(M)$. Formally, we may express the previous rules as follows:

$$S \in \Sigma(M) \stackrel{\text{def}}{=} \forall \alpha, \beta \in \mathit{Actions} \quad \begin{cases} \text{INIT} \in S \\ \alpha \in S \Rightarrow \alpha \in K \\ \alpha \in S \wedge \alpha \neq \text{INIT} \Rightarrow \text{INIT} <_S \alpha \\ (\alpha \rightarrow \beta) \wedge \alpha, \beta \in S \Rightarrow \alpha <_S \beta \\ (\alpha \lhd \beta) \Rightarrow (\beta \in S \Rightarrow \alpha \in S) \end{cases}$$

A site's current state is the *site-schedule* $S_A(t)$, which is a schedule $\in \Sigma(M_A(t))$. In case $S_A(t)$ has more than one schedule, we assume the site chooses one of the schedules according to some policy, which is out of the scope of our work.

We illustrate with an example from [SBS07]. Consider a database system (more precisely, a serializable database that transmits transactions by value). Let us assume that shared variables x, y, z start as zero. Two concurrent transactions $T_1 = r(x)0; w(z)1$ and $T_2 = w(x)2$ are related by $T_1 \rightarrow T_2$, since T_1 reads a value that precedes T_2's write.[12] T_1 and $T_3 = r(z)0; w(x)3$ are antagonistic, thus $T_1 \rightarrow T_3$ and $T_3 \rightarrow T_1$. In an execution where T_1 completes and then $T_4 = r(z)1$ runs, the latter transaction depends causally on the former, i.e., they may run only in that order, and T_4 aborts if T_1 aborts; we write $T_1 \rightarrow T_4 \wedge T_1 \lhd T_4$.

The third constraint type is Non-commutativity. It requires that, given each pair of non-commuting actions, the sound schedules resulting from every site-multilog must eventually order the action pair similarly. More precisely, when $\alpha \nparallel \beta$, the system must put either a $\alpha \rightarrow \beta$ or $\beta \rightarrow \alpha$ at every site-multilog.

Note that schedules may be sound but not satisfy Non-commutativity. In other words, Non-commutativity is a liveness obligation, and is irrelevant to schedule soundness.

As an example, transactions T_1 and $T_5 = r(y)0$ commute if x, y and z are independent. In a database system that commits operations (as opposed to committing values), we may execute transactions $T_6 =$"Credit 66 euros to Account 12345" and $T_7 =$"Credit 77 euros to Account 12345" in any order, as addition is a commutative operation. However, consider that applications require that the order in which users see the committed deposits be the same, no matter which site each user uses to access the account; in other words, that T_6 and T_7 are non-commutative. Then,

[10] A constraint is a relation in $\times$. By abuse of notation, for some relation $\mathcal{R}$, we write equivalently $(\alpha \mathcal{R} \beta) \in M$ or $\alpha \mathcal{R} \beta$ or $(\alpha, \beta) \in \mathcal{R}$. $\nparallel$ is symmetric and $\lhd$ is reflexive. They do not have any further special properties; in particular, $\rightarrow$ and $\lhd$ are not transitive, are not orders, and may be cyclic.

[11] We use ACF terminology for coherence.

[12] $r(x)n$ stands for a read of x returning value n, and $w(x)n$ writes the value n into x.

Algorithm 12 *ClientActionsConstraints*(L)

Require: $L \subseteq Actions$
1: $K_A := K_A \cup L$
2: **for** all $(\alpha,\beta) \in K_A \times K_A$ such that $\alpha \rightarrow_{\mathcal{M}} \beta$ **do**
3: $\quad \rightarrow_A := \rightarrow_A \cup \{(\alpha,\beta)\}$
4: **for** all $(\alpha,\beta) \in K_A \times K_A$ such that $\alpha \lhd_{\mathcal{M}} \beta$ **do**
5: $\quad \lhd_A := \lhd_A \cup \{(\alpha,\beta)\}$
6: **for** all $(\alpha,\beta) \in K_A \times K_A$ such that $\alpha \nparallel_{\mathcal{M}} \beta$ **do**
7: $\quad \nparallel_A := \nparallel_A \cup \{(\alpha,\beta)\}$

each site may commit T_6 and T_7 in any order, as long as that order is the same at all sites.

4.3.2 Optimistic Replication with the ACF

We now describe a generic optimistic replication protocol using the ACF formalism.

4.3.2.1 Client Behaviour and Client Interaction

A client performs tentative operations by submitting actions and constraints to its local site-multilog. Actions and constraints will eventually propagate to all sites by a protocol such as the one we present in Section 3.2.6. As the client submits a set of actions, L, they are added to its site-multilog. Complementarily, the client also adds the constraints that exist between the actions in L, as well as between the actions in L and the actions already present in the site-multilog.

We assume that, given a pair of actions, α and β, any client will determine the same set of constraints between α and β. This is a realistic assumption, generally present in semantic optimistic replication systems; e.g. Bayou's conflict checks are deterministic functions, which always produce the same output when applied upon the same update schedule.

We abstract such a property of constraint determination by postulating that clients have access to a sound multilog containing all the semantic constraints between any possible action that may be submitted: $\mathcal{M} = (A, \rightarrow_{\mathcal{M}}, \lhd_{\mathcal{M}}, \nparallel_{\mathcal{M}})$. As the client adds actions to its site-mutilog, function *ClientActionsConstraints* (Algorithm 12) adds constraints with respect to actions that the site already knows.[13]

To illustrate, consider Alice and Bob working together. Alice uses their shared calendar at Site *A*, and Bob at Site *B*. Planning a meeting with Bob in Paris, Alice submits two actions: α ="Buy train ticket to Paris next Monday at 10:00" and β ="Attend meeting". As β depends causally on α, $\mathcal{M}$ contains $\alpha \rightarrow_{\mathcal{M}} \beta$ and $\alpha \lhd_{\mathcal{M}} \beta$. Alice calls *ClientActionsConstraints*$(\{\alpha\})$ to add action α to site-multilog

[13] In the pseudo-code, we leave the current time t implicit. A double-slash and sans-serif font indicates a comment, as in // This is a comment.

Algorithm 13 *ReceiveAndCompare*(M)

Declare: $M = (K, \rightarrow, \lhd, \nparallel)$ a multilog that site A receives from a remote site
  $M_A := M_A \cup M$
  for all $(\alpha, \beta) \in K_A \times K_A$ such that $\alpha \rightarrow_{\mathcal{M}} \beta$ **do**
    $\rightarrow_A := \rightarrow_A \cup \{(\alpha, \beta)\}$
  for all $(\alpha, \beta) \in K_A \times K_A$ such that $\alpha \nparallel_{\mathcal{M}} \beta$ **do**
    $\nparallel_A := \nparallel_A \cup \{(\alpha, \beta)\}$

M_A, and, some time later, similarly for β. At this point, Algorithm 12 adds the constraints $\alpha \rightarrow \beta$ and $\alpha \lhd \beta$ taken from $\mathcal{M}$.

4.3.2.2 Multilog Propagation

The new actions and constraints that a client adds form a multilog that is sent to remote sites during synchronization sessions with such sites. Upon reception, receivers merge this multilog into their own site-multilog. By anti-entropy, every site eventually receives all actions and constraints submitted at any site.

When Site A receives a multilog M, it executes function *ReceiveAndCompare* (Algorithm 13), which first merges what it received into the local site-multilog. Then, if any constraints exist between previously-known actions and the received ones, it adds the corresponding constraints to the site-multilog.[14]

Let us return to Alice and Bob. Suppose that Bob now adds action γ, meaning "Cancel the meeting," to M_B. Action γ is antagonistic with action β; hence, $\beta \rightarrow_{\mathcal{M}} \gamma \wedge \gamma \rightarrow_{\mathcal{M}} \beta$. Some time later, Site B sends its site-multilog to Site A; when Site A receives it, it runs Algorithm 13, notices the antagonism, and adds constraint $\beta \rightarrow \gamma \wedge \gamma \rightarrow \beta$ to M_A. Thereafter, site-schedules at Site A may include either β or γ, but not both.

4.3.2.3 Commitment

Epidemic communication ensures that all site-multilogs eventually receive all information, but site-schedules might still differ between sites.

For instance, let us return to Alice and Bob. Assuming users add no more actions, eventually all site-multilogs become $(\{\text{INIT}, \alpha, \beta, \gamma\}, \{\alpha \rightarrow \beta, \beta \rightarrow \gamma, \gamma \rightarrow \beta\}, \{\alpha \lhd \beta\}, \varnothing)$. In this state, actions remain tentative; at time t, Site A might execute $S_A(t) = \text{INIT}; \alpha; \beta$, Site B $S_B(t) = \text{INIT}; \alpha; \gamma$, and just INIT at $t+1$. A commitment protocol ensures that α, β and γ eventually stabilize, and that both Alice and Bob learn the same outcome. For instance, the commitment protocol might add $\beta \lhd \text{INIT}$ to M_A, which guarantees β, thereby both guaranteeing α and killing γ. α, β and γ are now decided and stable at Site A. M_A eventually propagates to

[14] *ClientActionsConstraints* provides constraints between successive actions submitted at the same site. These consist typically of dependence and atomicity constraints. In contrast, *ReceiveAndCompare* computes constraints between independently-submitted actions.

other sites; and inevitably, all site-schedules eventually start with INIT;α;β, and γ is dead everywhere.

Before proposing solutions for the commitment problem, we formalize it using the ACF. In the ACF, an action executes tentatively only, because of conflicts and related issues. However, an action might have sufficient constraints that its execution is stable. The ACF distinguishes the following interesting subsets of actions relative to M.

- *Guaranteed* actions appear in every schedule of $\Sigma(M)$. By definition, INIT is always guaranteed. Formally, $Guar(M)$ is the smallest subset of K satisfying: $\text{INIT} \in Guar(M) \wedge ((\alpha \in Guar(M) \wedge \beta \lhd \alpha) \Rightarrow \beta \in Guar(M))$.

- *Dead* actions never appear in a schedule of $\Sigma(M)$. $Dead(M)$ is the smallest subset of *Actions* satisfying: $((\alpha_1, \ldots, \alpha_{m\geq 0} \in Guar(M)) \wedge (\beta \rightarrow \alpha_1 \rightarrow \ldots \rightarrow \alpha_m \rightarrow \beta) \Rightarrow \beta \in Dead(M)) \wedge ((\alpha \in Dead(M) \wedge \alpha \lhd \beta) \Rightarrow \beta \in Dead(M))$.

- *Serialized* actions are either dead or ordered with respect to all Non-Commutativity constraints.
  $Serialised(M) \stackrel{\text{def}}{=} \{\alpha \in K | \forall \beta \in K, \quad \alpha \not\parallel \beta \Rightarrow \alpha \rightarrow \beta \vee \beta \rightarrow \alpha \vee \beta \in Dead(M) \vee \alpha \in Dead(M)\}$.

- *Decided* actions are either dead, or both guaranteed and serialized.
  $Decided(M) \stackrel{\text{def}}{=} Dead(M) \cup (Guar(M) \cap Serialised(M))$.

- *Stable* (i.e., *durable*) actions are decided, and all actions that precede them by NotAfter or Enables are themselves stable: $Stable(M) \stackrel{\text{def}}{=} Dead(M) \cup \{\alpha \in Guar(M) \cap Serialised(M) | \forall \beta \in Actions \quad (\beta \rightarrow \alpha \vee \beta \lhd \alpha) \Rightarrow \beta \in Stable(M)\}$.

Using the terminology introduced in Section 3.2, a *committed action* corresponds to one that is both guaranteed and stable. Similarly, an *aborted action* corresponds to a dead action.

The commitment protocol is responsible for deciding actions. To *decide* an action α, relative to a multilog M, means to add constraints to M such that $\alpha \in Decided(M)$. More precisely:

- To guarantee α, we add $\alpha \lhd$ INIT to the multilog. Since INIT is, by definition, guaranteed, this implies that must also be guaranteed.

- To kill α, we add $\alpha \rightarrow \alpha$. This makes it impossible for α to appear in any sound schedule.

- To serialize non-commuting actions α and β, we add either $\alpha \rightarrow \beta$, $\beta \rightarrow \alpha$, $\alpha \rightarrow \alpha$, or $\beta \rightarrow \beta$. This either imposes an order between both actions, which eventually becomes reflected at every site-multilog, or aborts one of the actions.

α	$<$	β	$<$	γ	Decision	
		β	$\nparallel$	γ	(Serialise)	$\beta \rightarrow \gamma$
guar.	$\leftarrow$	β			(Kill β)	$\beta \rightarrow \beta$
dead	$\lhd$	β			(β is dead)	
		β	$\rhd$	γ	(Kill β)	$\beta \rightarrow \beta$
β not dead by above rules					(Guarantee β)	$\beta \lhd \text{INIT}$

Figure 4.10: $\mathcal{A}_{\text{Conservative}(<)}$: Applying semantic constraints to a given total order

Multilog M is said sound iff $\Sigma(M) \neq \varnothing$, or equivalently, iff $Dead(M) \cap Guar(M)$ is empty. An unsound multilog is definitely broken, i.e., no possible schedule can satisfy all the constraints, not even the empty schedule.

We are interested in a solution that achieves Explicit Eventual Consistency, as defined in Section 3.2. The ACF [SB04] defines a related property which, in ACF terminology, we express by the following conditions:

Definition 16. *Eventual Consistency in the ACF. An optimistically replicated system is eventually consistent iff it satisfies all the following conditions:*

- *Local soundness (safety): Every site-schedule is sound:*

$$\forall A, t \quad S_A(t) \in \Sigma(M_A(t))$$

- *Mergeability (safety): The union of all the site-multilogs over time is sound:*

$$\Sigma(\bigcup_{A,t} M_A(t)) \neq \varnothing$$

- *Eventual propagation (liveness):* $\forall A, B \in Sites \quad \forall t \quad \exists t' : M_A(t) \subseteq M_B(t')$

- *Eventual decision (liveness): Every submitted action is eventually decided:*

$$\forall \alpha \in Actions \quad \forall A \in Sites \quad \forall t \quad \exists t' : K_A(t) \subseteq Decided(M_A(t'))$$

Local soundness guarantees that every execution is safe. Mergeability ensures that, globally, the system remains sound. Eventual propagation ensures that information available at some site is eventually known everywhere. Eventual decision guarantees that every action is eventually decided.

A commitment algorithm aims to fulfill the obligations of Eventual Decision. Of course, it must also satisfy the safety requirements. The next sections describe solutions for such a problem. We start by easier centralized solutions, and then we depart to a more interesting decentralized solution.

4.3.3 Centralized Semantic Commitment

Previously, existing solutions that solve commitment while taking into account a rich semantic repertoire as the ACF's were centralized. We denote $\mathcal{A}(M)$ any such centralized algorithm. $\mathcal{A}(M)$ offers decisions based on a single multilog M (the site-multilog of a distinguished site).

Assuming M is sound, and noting the result $M' = \mathcal{A}(M)$, $\mathcal{A}$ must satisfy the following requirements:

- $\mathcal{A}$ extends its input: $M \subseteq M'$.
- $\mathcal{A}$ may not add actions: $K' = K$.
- $\mathcal{A}$ may add constraints, which are restricted to decisions:

$$\begin{array}{rcl} \alpha \rightarrow' \beta & \Rightarrow & (\alpha \rightarrow \beta) \vee (\alpha \nparallel \beta) \vee (\beta = \alpha) \\ \alpha \lhd' \beta & \Rightarrow & (\alpha \lhd \beta) \vee (\beta = \text{INIT}) \\ \nparallel' & = & \nparallel \end{array}$$

- M' is sound.
- M' is stable: $\mathit{Stable}(M') = K$.

$\mathcal{A}$ could be any algorithm satisfying the above requirements.

One possible algorithm, which we denote $\mathcal{A}_{\text{Conservative}(<)}$, starts by ordering actions, then kills actions for which the order is unsafe. Figure 4.10 illustrates its operation. Assume that the $\mathcal{A}_{\text{Conservative}(<)}$ has determined a total order, $<$, of the actions in a a sound multilog, M. The algorithm decides one action at at time, iterating over all actions, following the previous order. Let us call the current action β. Consider actions α and γ such that $\alpha < \beta < \gamma$: α has already been decided, while γ has not. If $\beta \nparallel \gamma$, then $\mathcal{A}_{\text{Conservative}(<)}$ serializes them in schedule order. If $\beta \rightarrow \alpha$, and α is guaranteed, then $\mathcal{A}_{\text{Conservative}(<)}$ kills β, because β is incompatible with the previous decisions. If $\gamma \lhd \beta$, $\mathcal{A}_{\text{Conservative}(<)}$ kills β; in this case, it is not known whether γ can be guaranteed, so $\mathcal{A}_{\text{Conservative}(<)}$ takes a conservative option. By definition, if $\alpha \lhd \beta$ and α is dead, then β is dead. If β is not dead by any of the above rules, then $\mathcal{A}_{\text{Conservative}(<)}$ adds $\beta \lhd \text{INIT}$ to the multilog, thus deciding β guaranteed. The resulting $\Sigma(\mathcal{A}_{\text{Conservative}(<)}(M))$ contains a unique schedule.

This approach is safe. However, it should be clear that it tends to kill actions unnecessarily. The Bayou system [TTP+95] applies $\mathcal{A}_{\text{Conservative}(<)}$, where $<$ is the order in which actions are received at a single primary site. An action aborts if it fails its dependency check fails (and the corresponding merge procedure does nothing); we can reify this case as a $\rightarrow$ constraint.

An alternative to $\mathcal{A}_{\text{Conservative}(<)}$ is that of IceCube system [KRSD01]. $\mathcal{A}_{\text{IceCube}}$ does not compute an arbitrary order on actions, rather tries to minimize the number of dead actions in $\mathcal{A}_{\text{IceCube}}(M)$. It does so by an optimization algorithm, which heuristically compares all possible sound schedules that can be generated from the current site-multilog. The system then outputs a set of possible decisions to the user, who chooses which one to follow.

4.3.4 A Generic Decentralized Commitment Protocol

We now address decentralized approaches for semantic commitment. To decentralize commitment, one approach might be to determine a global total order $<$, using a decentralized consensus algorithm such as Paxos [Lam98], and to apply $\mathcal{A}_{\text{Conservative}(<)}$. As above, this order is arbitrary and $\mathcal{A}_{\text{Conservative}(<)}$ tends to kill unnecessarily.

Instead, the algorithm proposed here allows each site to propose decisions that minimize aborts and to reach consensus on these proposals in a decentralized manner. This is the subject of the next sections.

4.3.4.1 Overview

Intuitively, one can regard eventual consistency as equivalent to the property that the site-multilogs of all sites share a common *well-formed subset* (which we define hereafter) of stable actions, which grows to include every action eventually. Commitment tries to agree on subsequent extensions of this subset; while, in the meantime, clients continue to make optimistic progress beyond this subset.

In our protocol, different sites run instances of $\mathcal{A}$ to make proposals for decision of the actions in their site-multilogs. A proposal is a subset of the site-multilog in which all its actions are decided. Sites incrementally agree on subsets of proposals, called *eligible candidates*, using an epidemic weighted voting protocol. This works even if $\mathcal{A}$ is non-deterministic, or if sites use different $\mathcal{A}$ algorithms. Any algorithm that satisfies the requirements of Section 4.3.3 is suitable.

In what follows, i represents the current site, and j,k range over *Sites*.

We distinguish two roles at each site, proposers and acceptors. A proposer at some site chooses candidate decisions upon the actions in the corresponding site-multilog. We employ an epidemic voting approach for commitment, similarly to VVWV. Each proposer has a fixed *weight*, such that $\sum_{A \in Sites} weight_A = 1$. For presentation simplicity, and without loss of generality, weights are distributed ahead of time and remain static.

An acceptor at some site computes the outcome of an election once it becomes aware of enough votes, and inserts the corresponding decision constraints into the local site-multilog.

Each site A stores the most recent proposal received from each proposer in array $proposals_A$, of size $n = |Sites|$ (the number of sites). To keep track of proposals, each entry $proposals_A[B]$ carries a logical timestamp, denoted $proposals_A[B].ts$. Timestamping ensures allows eventual dissemination of proposals even when network links are not FIFO.

Algorithm 14 determines the behavior of each site. First it initializes the site-multilog and proposals, then it runs a number of parallel iterative threads, which we detail next. Within a thread, an iteration is atomic, and iterations are separated by arbitrary amounts of time.

Algorithm 14 Algorithm at Site A (each parallel loop is assumed atomic)

Declare: M_A: local site-multilog
Declare: $proposals_A[n]$: array of proposals, indexed by site; a proposal is a multilog

1: $M_A := (\{\text{INIT}\}, \varnothing, \varnothing, \varnothing)$
2: $proposals_A := [((\{\text{INIT}\}, \varnothing, \varnothing, \varnothing), 0), \ldots, ((\{\text{INIT}\}, \varnothing, \varnothing, \varnothing), 0)]$
3: **loop** // Epidemic transmission
4: Choose $B \neq A$;
5: Send copy of M_A and $proposals_A$ to B
6: ||
7: **loop** // Epidemic reception
8: Receive multilog M and proposals P from some site $B \neq A$
9: $ReceiveAndCompare(M)$ // Compute conflict constraints
10: $MergeProposals(P)$
11: ||
12: **loop** // Client submits
13: Choose $L \subseteq Actions$
14: $ClientActionsConstraints(L)$ // Submit actions, compute local constraints
15: ||
16: **loop** // Compute current local state
17: Choose $S_A \in \Sigma(M_A)$
18: Execute S_A
19: ||
20: **loop** // Proposer
21: $UpdateProposal$ // Suppress redundant parts
22: $proposals_A[A] := \mathcal{A}(M_A \cup proposals_A[A])$ // New proposal, keeping previous
23: Increment $proposals_A[A].ts$
24: ||
25: **loop** // Acceptor
26: $w := Elect()$
27: $M_A := M_A \cup w$

4.3.4.2 Epidemic Communication

The first two threads (lines 3–10) exchange multilogs and proposals between sites. Function *ReceiveAndCompare* (defined in Algorithm 13, Section 4.3.2.2) performs conflict detection, comparing newly received actions to previous ones, hence computing constraints among the former and the latter. In Algorithm 16 a receiver updates its own set of proposals with more recent ones.

Algorithm 15 *UpdateProposal*

1: Let $P = (K_P, \rightarrow_P, \lhd_P, \not\parallel_P) = proposals_A[A]$
2: $K_P := K_P \setminus Decided(M_A)$
3: $\rightarrow_P := \rightarrow_P \cap K_P \times K_P$
4: $\lhd_P := \lhd_P \cap K_P \times K_P$
5: $\not\parallel_P := \varnothing$
6: $proposals_A[A] := P$

Algorithm 16 *MergeProposals*(P)

1: **for all** $B \in Sites$ **do**
2: **if** $proposals_A[B].ts < P[B].ts$ **then**
3: $proposals_A[B] := P[B]$
4: $proposals_A[B].ts := P[B].ts$

4.3.4.3 Client, Local State, Proposer

The third thread (lines 12–14) handles tentative activity that the local users submit. An application submits tentative operations to its local site-multilog, which the site-schedule will try to execute in the fourth thread. Constraints relating new actions to previous ones are included at this stage by function *ClientActionsConstraints* (which Algorithm 12 defines).

The fourth thread (lines 16–18) computes the current tentative state by selecting and executing some sound site-schedule.

The fifth thread (20–23) computes proposals by invoking $\mathcal{A}$. A proposal extends the current site-multilog with proposed decisions. A proposer may not retract a proposal that it has already proposed (as it may already have been received by some other site). Passing argument $M_A \cup proposals_A[A]$ to $\mathcal{A}$ ensures these two conditions. However, once a candidate has either won or lost an election, it becomes redundant; *UpdateProposal* removes it from the proposal (Algorithm 15).

The last thread (25–27) conducts elections. It waits for eligible candidates (taken from the proposals that the the site knows of) to be elected. When a site determines that an eligible candidate, *w*, has collected enough votes to be elected, it adds the actions and constraints of *w* to the site-multilog. Since the actions in *w* are decided actions, they also become decided in the site-multilog. This achieves the goal of the commitment protocol.

The next section describes the election decision algorithm (*Elect*).

4.3.5 Election Decision with Semantics

Election decision constitutes the crucial and most difficult step of our decentralized semantic commitment protocol. We start by describing this step in abstract, exposing the requirements for election safety. We then introduce two particular solutions and discuss their appropriateness.

Essentially, we use the same approach as VVWV, i.e., epidemic weighted voting. Each proposal (cast by a proposer) is a multilog that possibly includes multiple candidates, each consisting of a different sub-multilog of the proposal that contains decided actions only. From such candidates, only some well-formed candidates are *eligible candidates*. For the moment we leave the notion of eligible candidates undefined; we return to it at the end of the section.

Proposers vote for proposals. A vote for a proposal means a vote for each eligible candidate in the proposal's multilog. As with VVWV, the election decision is taken locally at each site, exclusively using local knowledge. Intuitively, an eligible candidate is elected once it collects enough votes to win the election no matter the votes that any other *incompatible* eligible candidate (that some other proposer may propose) may obtain (according to a particular epidemic coterie). As with VVWV, we consider the epidemic plurality coterie, $\mathcal{LP}$ (see Section 4.1.1.2).

As votes become locally available, the election decision algorithm checks whether any eligible candidate from one of the known proposals satisfies the election condition. We formalize the election condition at a given site A as follows.

Definition 17. *Election Condition.*

Let w be an eligible candidate such that $\exists B : w \subseteq proposals_A[B]$. w wins an election when, for all eligible candidate x such that x and w are incompatible:

1. $voted_A(w) > voted_A(x) + collectable_A(x)$, *or*
2. $voted_A(w) = voted_A(x) + collectable_A(x)$ *and* $w \prec_A x$.

$\prec_A$ *is the tie-breaking rule, as defined in Section 4.2.6.1.*

The above condition is practically identical[15] to the election condition of VVWV (see Section 4.2.5). In fact, the intuition behind the condition is similar. Nevertheless, the definitions of *incompatible candidates*, *voted* and *collectable* are now defined in terms of the ACF, as we describe next. The notion of compatibility is now more general than in VVWV, as it takes into account the rich semantics contained in the multilogs.

Definition 18. *Two multilogs M and M' are compatible (which we denote by $compatible(M,M')$) when their union is sound: $compatible(M,M') \stackrel{\text{def}}{=} \Sigma(M \cup M') \neq \varnothing$. Otherwise, we say M and M' are incompatible.*

The votes for some eligible candidate, c, are now given by the votes for proposals containing the candidate. A proposal that votes for an eligible candidate does not vote for any other incompatible eligible candidate; otherwise, the proposal

[15] In fact, it leaves out the condition of w having more than 50% of votes. VVWV needs such a condition in order to deal with votes equal to $\perp$. Since our description of the semantic algorithm does not allow $\perp$ proposals, such a condition becomes obsolete. Notice that we consider $\perp$-votes in VVWV for the sake of presentation simplicity, only; VVWV may also be described without $\perp$-votes and using the above election condition.

would not be sound. Furthermore, as long as none of c's incompatible candidates becomes elected, a proposal voting for c will always hold on to such a vote; this is easily proven by noting that, in such a case, the proposal can only evolve monotonically by addition of new actions and constraints. Hence, the following definition of *voted* captures an equivalent meaning as the homonymous definition in VVWV: as long as none of c's incompatible eligible candidates is elected, a vote for c will always support c and never support any of its incompatible eligible candidates, no matter how the vote evolves.

Definition 19. *The votes for a given eligible candidate, c, at a site A, denoted $voted_A(c)$, are given by:*

$$voted_A(c) \stackrel{\text{def}}{=} \sum_{B:c \subseteq proposals_A[B]} weight_B$$

Finally, we need to define collectable votes in the semantic context. Clearly, a proposal that does not yet vote for an eligible candidate, c, but is still compatible with c may potentially grow to include c. For that, it just suffices to add the missing actions and constraints to the proposal. If, otherwise, we consider an incompatible proposal, it will never grow to include c; otherwise, the resulting proposal would not be sound. Consequently, the collectable votes of c consist of the votes for any proposal that, not including c, are compatible with c.

Definition 20.

$$collectable_A(c) \stackrel{\text{def}}{=} \sum_{B:c \not\subseteq proposals_A[B] \wedge compatible(c, proposals_A[B])} weight_B$$

We return to our example of Alice and Bob. Recall that, once Alice and Bob have submitted their actions, and A and B have exchanged site-multilogs, both site-multilogs are equal to $(\{\text{INIT}, \alpha, \beta\}, \{\alpha \rightarrow \beta, \alpha \rightarrow \gamma, \gamma \rightarrow \alpha\}, \{\alpha \lhd \beta\}, \varnothing)$. Now Alice (Site A) proposes to guarantee α and β, and to kill γ: $proposals_A[A] = M_A \cup \{\beta \lhd \text{INIT}\}$. In the meanwhile, Bob at Site B proposes to guarantee γ and α, and to kill β: $proposals_B[B] = M_B \cup \{\gamma \lhd \text{INIT}, \alpha \lhd \text{INIT}\}$. These proposals are incompatible; therefore the commitment protocol will eventually agree on at most one of them.

Consider now a third site, Site C; assume that the three sites have equal weight $\frac{1}{3}$. Imagine that Site C receives Site B's site-multilog and proposal, and sets its own proposal to one identical to Site A's. Some time later, Site C sends its proposal to Site A. At this point, Site A has received all sites' proposals. Now Site A might run an election, considering a candidate X equal to $proposals_A[A]$.

Now suppose that X is eligible. $voted_A(X) = \frac{2}{3}$ is greater than the votes that any incompatible eligible candidate may potentially collect, which is $\frac{1}{3}$. Therefore, Site A elects X and merges X into M_A. Any other site will either elect X (or some compatible candidate) or become aware of its election by epidemic transmission of M_A.

Evidently, the safety of election decisions requires that no two incompatible eligible candidates be elected at any two sites. Interestingly, it is key to such a requirement that the definition of eligible candidate be sufficiently strict.

We illustrate with an example. Let us naively define eligible candidate as *any* subset of a proposal; i.e., any multilog would be an eligible candidate. Now, consider that three different sites, *A*, *B* and *C*, each concurrently issued action α, β and γ, respectively. Further, let their semantics be such that $\alpha \rightarrow \beta$, $\beta \rightarrow \gamma$, and $\gamma \rightarrow \alpha$; in other words, the commitment protocol must abort (at least) one of the actions, in order to break the cycle.

Assume that, by anti-entropy, every (undecided) action propagate to every site-multilog. Further, assume that *A* proposes to guarantee α and β; *B* proposes to guarantee β and γ; while *C* proposes to guarantee α and γ. When *A* receives *C*'s proposal, it learns that the eligible candidate corresponding to "α *guaranteed*" ($\alpha \lhd \text{INIT}$) has collected $\frac{2}{3}$ of votes, which is enough to elect it. However, if *B* receives *A*'s proposal, and *C* receives *B*'s proposal, *B* and *C* will also determine that the eligible candidates ´´β *guaranteed*" and ´´γ *guaranteed*" (respectively) have collected enough votes for election. Hence, this counter-example results in each site having guaranteed one of the three actions. This clearly violates the mergeability property of eventual consistency, since at most two of α, β and γ can be guaranteed.

Intuitively, we can solve the problem exposed above by requiring that eligible candidates be larger multilogs. Nevertheless, there is an inherent trade-off in the definition of eligible candidates, because we also want eligible to be as small as possible. Allowing small eligible candidates means that we may earlier consider them for eligibility (and, hence, commitment). For instance, if a given sub-multilog of a proposal (held at some site) does not contain all the actions and constraints that are required for it to be an eligible candidate, then the site must wait to receive such elements from other sites.

Furthermore, a smaller eligible candidate will collect votes faster, since a higher number of proposals will include the smaller candidate multilog, in comparison to a superset of it. For instance, it is easy to see that, in the above example, small eligible candidates of one action would receive more votes than in the case where eligible candidates had to include all the actions of each cycle of NotAfter constraints.

The next sections describe and discuss two safe definitions of eligible candidates.

4.3.5.1 Solution 1: Happens-Before-Related Eligible Candidates

To illustrate a safe definition of eligible candidate, we start by devising one such definition from VVWV. As Section 4.2 explains, the sole semantics that VVWV considers about its actions are derived from the happens-before relation between them. More precisely, if two actions, α and β, are related by happens-before, then VVWV assumes them to be causally related; if, otherwise, the actions are

concurrent by happens-before, VVWV regards them as antagonistic. Using ACF's terminology, this means that the universe of multilogs in VVWV is restricted to those constructed as follows.

Given a pair of actions, α and β:

- If α happens-before β (or vice-versa), there exists a possibly empty (happens-before-related) sequence of actions, $\gamma_1, .. \gamma_n$, such that $\alpha \rightarrow \gamma_1 \wedge \alpha \lhd \gamma_1 \wedge .. \wedge \gamma_n \rightarrow \beta \wedge \gamma_n \lhd \beta$ (or, respectively, $\beta \rightarrow \gamma_1 \wedge \beta \lhd \gamma_1 \wedge .. \wedge \gamma_n \rightarrow \alpha \wedge \gamma_n \lhd \alpha$);
- If α and β are concurrent by happens-before, then $\alpha \rightarrow \beta$ and $\beta \rightarrow \alpha$.
- No more client constraints exist between α and β.

The eligible candidates are subsets of the above multilogs. Each eligible candidate includes a given action and all the actions that causally precede it, all of which are guaranteed. In other words, an eligible candidate tries to commit a sequence of happens-before-related actions (starting from the initial state), as Section 4.2.5 defined.

Using this definition of eligible candidates, we obtain VVWV. The safety of VVWV implies that this definition of eligible candidates is safe. However, in practice this definition assumes a semantic-oblivious protocol, in which multilogs cannot be more expressive than the construction we described above.

4.3.5.2 Solution 2: Stable-Closed Eligible Candidates

A more interesting eligible candidate definition would consider the full semantic power of the ACF, i.e. eligible candidates taken from the complete universe of sound multilogs. This section describes one such definition, as proposed by Sutra [SBS07], and proven safe by Sutra in [SBS06]. We designate it as *stable-closed eligible candidates*.

A stable-closed eligible candidate[16] is a multilog that satisfies two conditions. First, the multilog must be a *well-formed prefix* of some proposal *P*:

Definition 21. *Well-formed prefix. Let* $M = (K, \rightarrow, \lhd, \nparallel)$ *and* $M' = (K', \rightarrow', \lhd', \nparallel')$ *be two multilogs.* M' *is a* well-formed prefix *of* M *, noted* $M' \overset{wf}{\sqsubset} M$*, if (i) it is a subset of* M *, (ii) it is stable, (iii) it is left-closed for its actions, and (iv) it is closed*

[16] Stable-closed eligible candidates allow for a smaller set of incompatible eligible candidates to be considered in the election condition. More precisely, given a stable-closed eligible candidate *w*, the election condition for *w* needs only consider the incompatible eligible candidates that contain the same set of actions as *w*. A formal definition may be consulted in [SBS07].

for its constraints.

$$M' \overset{wf}{\sqsubset} M \overset{\text{def}}{=} \begin{cases} M' \subseteq M \\ K' = Stable(M') \\ \forall \alpha, \beta \in Actions \quad \beta \in K' \Rightarrow \begin{cases} \alpha \rightarrow \beta \Rightarrow \alpha \rightarrow' \beta \\ \alpha \lhd \beta \Rightarrow \alpha \lhd' \beta \\ \alpha \nparallel \beta \Rightarrow \alpha \nparallel' \beta \end{cases} \\ \forall \alpha, \beta \in Actions \quad (\alpha \rightarrow' \beta \vee \alpha \lhd' \beta \vee \alpha \nparallel' \beta) \Rightarrow \alpha, \beta \in K' \end{cases}$$

For instance, if a $\rightarrow$ or $\lhd$ cycle is present in M , every well-formed prefix either includes the whole cycle, or none of its actions.

Unfortunately, because of concurrency and asynchronous communication, it is possible that some sites know of a $\rightarrow$ cycle, while others do not. Moreover, some sites may only be aware of parts of a cycle. Therefore, we also need the following second condition for eligibility:

Definition 22. *Omniscience Condition. A multilog M satisfies the omniscience condition if, for any action in K, all the action's predecessors by client*[17] *NotAfter, Enables and NonCommuting constraints are in M.*

More precisely: omniscient$(M) \overset{\text{def}}{=} \forall \alpha, \beta \in Actions \times K \quad (\alpha \rightarrow_{\mathcal{M}} \beta \vee \alpha \nparallel_{\mathcal{M}} \beta \vee \alpha \lhd_{\mathcal{M}} \beta) \Rightarrow \alpha \in K$.

At the time a site issues an action α, any other site may concurrently be issuing actions that have constraints with α. Therefore, in order for a site to guarantee omniscience for a candidate containing α, the site must have previously learned about the multilogs of *all* sites in the system. This requirement introduces a fundamental limitation on fault tolerance. Let us divide the commitment of a given candidate in two consecutive phases, (i) ensuring its omniscience and (ii) electing the (eligible) candidate. Phase (i) halts in the presence of any fault; we discuss such a limitation in Section 4.3.6. Once omniscience is guaranteed and the eligible candidate runs for election, the phase (ii) may finally complete despite a higher number of faults (it may complete when only a majority of processes is available).

However, it is not sufficient for a site to learn about all the concurrent actions at a given moment in order to ensure omniscience of some candidate. The site must also ensure that, from that moment on, no more actions that affect omniscience of the given candidate are added by any site. It is out of the scope of this thesis to present a solution to implement omniscience. A possible implementation is outlined in [SSB06].

4.3.6 Summary and Discussion

The focus of the contribution of this section is on applications with rich semantics. Previous approaches to replication either do not support a sufficiently rich repertoire of semantics, or rely on a centralized primary commit protocol.

[17] A client constraint is a constraint that was added by the client, not the proposer.

This section describes complementary work to VVWV, seeking a decentralized extension of VVWV for semantically-rich systems. In contrast to Section 4.2, this section considers that the commitment protocol has access to a rich semantic information relating the updates that applications submit. We assume that such a semantic repertoire is expressed using a representative state-of-the-art formalism, the Actions-Constraints Framework (ACF), which reifies semantic relations as constraints.

The constraints restrict scheduling behavior of the system. More precisely: an atomic group of actions is either entirely guaranteed or entirely dead; a dependent action may execute only after those it depends on; if actions are antagonistic, they may not all become guaranteed, and the system must choose at least one and make it dead; if actions are non-commuting, the system must choose an execution order. Eventual consistency puts an obligation on the commitment protocol to resolve antagonism and non-commutativity relations, and to eventually execute equivalent schedules at all sites.

We propose an abstract commitment protocol that, having access to semantic knowledge as an ACF multilog, achieves eventual consistency using epidemic weighted voting. The protocol depends on an appropriate definition of eligible candidates, which we leave as an open direction for future work. As an exercise, we describe and discuss two possible definitions. The first definition, which represents VVWV, does not exploit the full semantic repertoire that the ACF may express; hence, it is not a semantically rich solution.

The second solution, stable-closed candidates, in turn, considers the full semantic repertoire of ACF. However, its omniscience condition requires that every site be accessible before committing any piece of tentative work. Therefore, the usefulness of the commitment protocol when instantiated with such a definition is arguable. More precisely, the advantages that it attains by exploiting semantics come at a cost in terms of fault tolerance that may be too high for most applications. As an example, consider two sites, temporarily in different partitions, that issue semantically-compatible updates. If we used the semantic protocol with stable-closed eligible candidates, none of the sites would be able to commit any update while the partitioning persists. In the environments that this thesis considers, partitioning constitutes the norm. If, instead, the system relied on the semantic-oblivious VVWV commitment protocol, then the site in the majority partition would normally be able to commit its work; as a drawback, such a commit would imply aborting the other concurrent, yet compatible, updates. We believe that, for many applications, the second alternative is preferable than the first one.

On one hand, the work this section describes is incomplete due to the absence of a more satisfactory definition of eligible candidates. On the other hand, we believe that it contributes to an eventual accomplishment of the goal of decentralized semantic commitment. More precisely, the abstract protocol presented previously reduces the problem to the simpler one of devising an appropriate definition of eligible candidate; i.e. one that is sufficiently strict to guarantee safety, and that, simultaneously, considers sufficiently small eligible candidates so as to allow rapid

and fault-tolerant commitment.

Chapter 5

Decoupled Update Commitment

This chapter describes how to complement our system of replicas with non-replicas acting as carriers of consistency packets. Replicas generate consistency packets reflecting their state. The size of a consistency packet should be of a lower order of magnitude than the information normally exchanged in anti-entropy. As consistency packets are delivered to non-replicas through ad-hoc encounters, and eventually to replicas, they should contribute to the progress of replica consistency. This requires extending the consistency protocol so that it becomes able to make *some* progress by exchange of a *sufficiently small* subset of the information that is handled in anti-entropy.

Conceptually, anti-entropy exchanges information for two basic functions, update propagation and update commitment. The latter ensures that updates are consistently committed at all replicas, by (*i*) proposing a tentative update schedule; (*ii*) detecting and resolving conflicts; and (*iii*) agreeing on a schedule to be consistently committed.[1] We propose that consistency packets contain only the meta-data necessary for the agreement step (*iii*). This follows from the observation that (a) we may complete the agreement on a schedule independently of the remaining phases; and, (b) if decoupled from such phases, agreement is possible by the exchange of some lightweight meta-representation of schedules instead of the updates themselves (plus some algorithm-specific control data). This information comprises a consistency packet. In most systems, such a consistency packet will be sufficiently small for our intents, as the next sections explain.

The remainder of the chapter is organized as follows. Section 5.1 introduces an abstraction of generic commitment protocols, which Section 5.2 extends so as to support consistency packet exchange. Section 5.3 discusses the attainable benefits of such an extension. Section 5.4 addresses the particular case of commitment protocols based on version vectors, which Section 5.5 illustrates with the example of VVWV. Section 5.6 addresses consistency packet management. Finally, Section 5.7 summarizes and discusses future work.

[1] Note that some protocols may omit some of these steps.

5.1 Base Protocol Abstraction

We start by recalling the replication algorithms from Chapter 3.2. Algorithm 17 presents a simplified, abstract algorithm that addresses the phases of update commitment with which we are mainly concerned in this chapter. Namely, such phases are: (*i*) scheduling, (*ii*) conflict detection and resolution, and (*iii*) agreement. We abstract such phases by the functions *constructSch* (representing phases *i* and *ii* combined), and *proposeSch* and *getNewDecidedSch* (phase *iii*). For the sake of generalization, we leave the update propagation component unspecified. All these phases make progress by information exchanged during anti-entropy.

Note that Algorithm 17 is a simplified combination of the abstract algorithms from Chapter 3.2. For presentation simplicity, and without lack of generality, it considers only one object.

Algorithm 17 Update Commitment at replica a (each loop iteration assumed atomic)

```
1: loop
2:    if new updates {u_1,..,u_m} exist then
3:       sch_a ← constructSch(sch_a, {u_1,..,u_m});
4:       proposeSch(sch_a);
5: ||
6: loop
7:    if sch_a ≠ stable_a(sch_a) then
8:       sch' ← getNewDecidedSch(sch_a);
9:       sch_a ← constructSch(sch', sch_a \ sch');
```

As sets of new updates, $\{u_1,..,u_m\}$, are either generated at a replica a, or received at a by anti-entropy from sch_b of another replica b, they are inserted into sch_a by *constructSch*, in some order and state (i.e., $\underline{u_i}$ or $\overline{u_i}$) that keeps sch_a sound. As new schedules sch_a are incrementally built, they may then be proposed for agreement. This is described in lines 3 and 4 of Algorithm 17.

At each replica a, the agreement algorithm incrementally decides, locally at each replica a, which updates in sch_a have become stable (at a), and at which order and state. Such a decision is output by *getNewDecidedSch*. Given an original schedule, s, as input, $getNewDecidedSch(s)$ returns, as soon as a new agreement is reached at a, a sound schedule, s', (a) that is completely stable at a (i.e. $stable_a(s) = s$), and (b) that suffixes the stable prefix of the original one (i.e. $stable_a(s)$ is a prefix of $stable_a(s')$). *getNewDecidedSch* is called iteratively at a replica a as long as there are still tentative updates in sch_a (line 8 of Algorithm 17).

Figure 5.1 gives an example of update commitment with the above abstraction, in a partitioned 3-replica system. It assumes a voting algorithm for agreement that requires two replicas to vote for a tentative schedule to decide it as stable. When replicas A and B become accessible, two anti-entropysessions enable the commitment of update u.

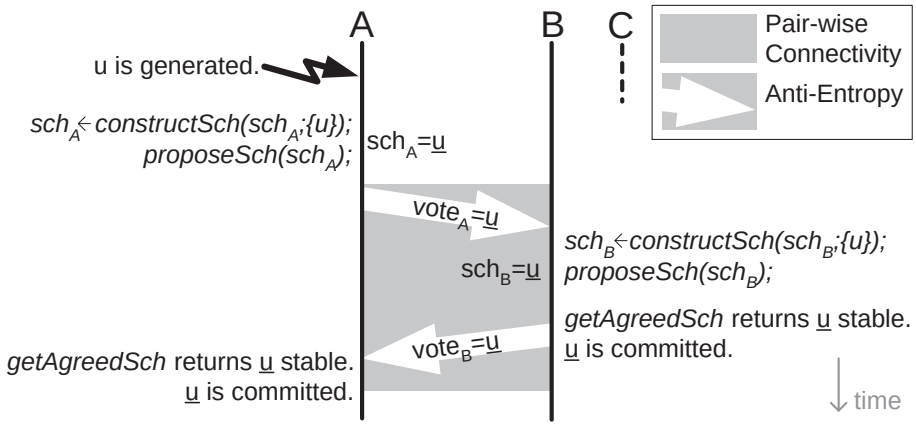


Figure 5.1: Example of update commitment.

5.2 Incorporating Consistency Packets

The above abstraction defines the agreement phase by functions (*proposeSch* and *getNewDecidedSch*) that manipulate updates included in schedules, plus some complementary control information specific of each particular algorithm.[2]

However, if we decouple the agreement phase from the remaining phases, we observe that proposed agreement algorithms (e.g. Bayou's primary-commit [PST+97] or VVWV's weighted voting [BF05b] agreement algorithms) do not actually require access to all the information an update holds; namely, they are not concerned with the op_u and $precond_u$ components of updates in a schedule. Instead, they may simply deal with some meta-data representation that identifies the (*i*) updates, (*ii*) their order, and (*iii*) their state (i.e. executed or aborted) in each schedule that is proposed or agreed. We call such a representation of a schedule *s* a *meta-schedule*, and denote it by *meta(s)*.

We designate the necessary information sent from one replica to another for the sole purpose of agreement as a *consistency packet*. It encapsulates meta-schedules as well as complementary control information of the agreement algorithm.

Hence, given a particular agreement algorithm, implemented by the *proposeSch* and *getNewDecidedSch* functions, one may directly obtain their meta-data versions. We designate them as *m-proposeSch* and *m-getNewDecidedSch*. Their input and output, respectively, are meta-schedules, instead of schedules. Whereas agreement by *proposeSch* and *getNewDecidedSch* progresses by anti-entropy, agreement by *m-proposeSch* and *m-getNewDecidedSch* evolves by exchange of consistency packets; these may be exchanged normally in the course of anti-entropy, or propagated individually through non-replicas.

A trivial meta-schedule consists of a list of update identifiers, id_{u_i}, ordered

[2]For instance, in weighted voting agreement (e.g. [BF05b]), such control information includes the weights and the replica identifier associated with each vote.

according to the corresponding schedule, and marked either as *executed* (denoted $\underline{id_{u_i}}$) or *aborted* (denoted $\overline{id_{u_i}}$). More efficient protocol-specific meta-schedules are also possible in some cases; the important case of version vectors is described in Section 5.4.

A meta-schedule is smaller in size than a schedule, as it no longer includes the op_u and $precond_u$ components of scheduled updates. For most systems, it will be of a lower order of magnitude in size than the corresponding schedule. We are interested in such systems. As a particular example, in the case of state-transfer systems [SS05] of reasonably sized objects, the size of schedules is dominated by the size of object state data (the op_u component); such a component is absent in meta-schedules. A second example is that of operation-transfer systems [SS05] with frequent updates; meta-schedule representations that do not depend on the number of updates (e.g. version vectors) will be significantly smaller in size than long schedules.

5.2.1 Protocol Extension

So far, we have a generic protocol that (*i*) fits into the abstraction defined in Section 5.1, relying on *proposeSch* and *getAgreed*, which manipulate and exchange schedules; but (*ii*) whose agreement algorithm may also work with meta-schedules, encapsulated in consistency packets (i.e. *m-proposeSch* and *m-getNewDecidedSch* exist).

We need to extend the protocol in order to use *m-proposeSch* and *m-getNewDecidedSch* instead of *proposeSch* and *getNewDecidedSch*, respectively. This section describes such an extension by proposing adaptation functions, *proposeSchEx* and *getNewDecidedSchEx* with a similar interface as *proposeSch* and *getNewDecidedSch*, respectively. They rely on *m-proposeSch* and *m-getNewDecidedSch*. Hence, the original protocol may be extended by transparently replacing proposeSch and *getNewDecidedSch* by *proposeSchEx* and *getNewDecidedSch*, respectively. The adaptation functions require the maintenance of a meta-schedule, $m\text{-}sch_r$, at each replica *a*. It represents the most recent stable meta-schedule that is known so far by *a*.

Algorithm 18 *proposeSchEx*(*schedule s*)

```
1: m-proposeSch(meta(s));
```

Algorithm 19 *schedule getNewDecidedSchEx*() (not assumed atomic)

```
1: Return the first of the following:
2: return commitOneLocalUpdate();
3: ||
4: m-sch_r ← m-getNewDecidedSch();
5: return commitOneLocalUpdate();
```

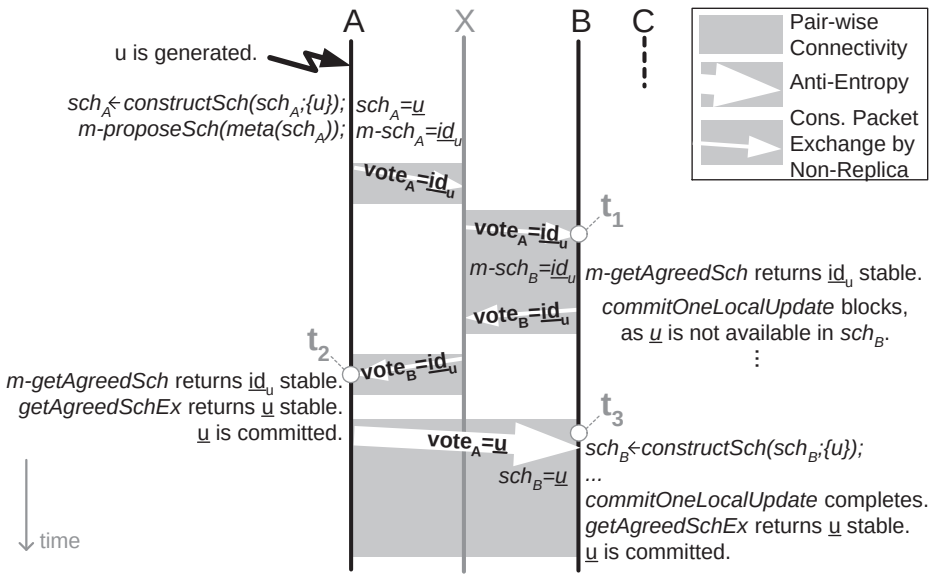


Figure 5.2: Example of update commitment with exchange of consistency packets.

Algorithm 20 *schedule commitOneLocalUpdate*() (not assumed atomic)

1: // Let $stable_a(sch_a) = u_1;..;u_k$, and
   $stable_a(m\text{-}sch_r) = id_{u_1};..;id_{u_k};..;id_{u_m}$ ($k \leq m$).
2: $s' \leftarrow stable_a(sch_a)$;
3: **wait until** $m > k$ and $u_{k+1} \in sch_a$;
4: **if** $\underline{id_{u_{k+1}}} \in m\text{-}sch_r$ **then**
5: $\quad s' \leftarrow stable_a(sch_a);\underline{u_{k+1}}$;
6: **else**
7: $\quad s' \leftarrow stable_a(sch_a);\overline{u_{k+1}}$;
8: mark u_{k+1} as stable in s';
9: return s';

The adaptation functions are convert schedules (manipulated by the original protocol) into meta-schedules (handled by the meta-data agreement versions), and vice-versa. Function *proposeSchEx*, presented in Algorithm 18, performs the first conversion; by definition, this is trivial. On the other hand, function *m-getNewDecidedSch*, described in Algorithms 19 and 20, converts meta-schedules into schedules. This conversion requires synchronization, since the updates identified by a meta-schedule may not yet be locally available in order to compose the corresponding schedule. In that case, the conversion waits for subsequent anti-entropy sessions until having received each needed update (Algorithm 20, line 3).

5.3 Benefits

The exchange of consistency packets via non-replicas adds an additional possibility of progress. Figure 5.2 illustrates that, continuing the example of Figure 5.1 with a passer-by non-replica, X. Instead of votes containing schedules, replicas A and B exchange consistency packets with X representing votes as meta-schedules. Despite being inaccessible for an initial period, A and B are able to exchange consistency packets through X. Consequently, agreement on a meta-schedule corresponding to $\underline{u}$ arrives before A and B could perform anti-entropy.

The *direct* effect of agreement acceleration does not, however, seem to be of great value to applications of the replicated system. However, it enables more interesting *indirect* improvements to the overall efficiency of the protocol. These are described as follows.

A. Commitment Acceleration. When agreement on a meta-schedule completes, the corresponding updates may be immediately committed if already available in sch_a (in a tentative form). Hence, agreement acceleration means commitment acceleration in this case. An example is that of replica A of Figure 5.2 (see t_2) after having received a consistency packet from B through X. Since update u is already available in sch_A, its actual commitment at A is also accelerated, even without anti-entropy.

B. Conflict Prevention. Accelerated agreement may cause the meta-schedule of a replica a to reference updates not yet included in sch_a. This denotes the existence of concurrent updates, whose awareness at a is anticipated by consistency packets. It implies that, if new updates are generated at a, there is a higher conflict probability than if no concurrent updates were known to exist. Applications may react to it by postponing planned writes to a time where the missing concurrent updates are received, hence reducing the conflict probability. An example is that of replica B in Figure 5.2, in moment t_1, after receiving a consistency packet from A denoting the existence of update u.

C. Reduction of *Hidden Conflicts*. Besides conflicts caused between updates generated concurrently at distinct replicas, conflicts may also arise when one update is generated *after* a another one has already been locally received. We designate this a *hidden conflict*.

To illustrate, let u_1 and u_2 be two updates received at replica a that are conflicting so that a sound schedule may only either include $\underline{u_1}$ and $\overline{u_2}$, or $\overline{u_1}$ and $\underline{u_2}$. Assume that sch_a includes $\underline{u_1}$ and $\overline{u_2}$, both still tentative. In this case, the tentative value of a, which applications access, will reflect the outcome of $\underline{u_1}$ but not of $\overline{u_2}$. Now, consider that an application issues an update u_3, added tentatively as $\underline{u_3}$ to sch_a. Given the natural assumption that issued updates are not conflicting with the value of the replica upon which they are issued, u_3 will be compatible with u_1; however, it may (or may not) conflict with u_2. Let us consider the case where such a conflict holds. Finally, assume that the agreement algorithm chooses a stable schedule with $\overline{u_1}$ and $\underline{u_2}$ (contrarily to the previous tentative schedule), and thus both commit at a. Accordingly, u_3 will abort ($\overline{u_3}$) in sch_a, due to a hidden conflict

with committed update $\underline{u_2}$.

Clearly, the agreement delay of a tentative schedule determines the time window during which hidden conflicts may arise. Hence, an accelerated agreement shortens such a window and, therefore, contributes to reducing the ratio of aborted updates.

D. Efficient Update Propagation. Agreement acceleration increases the frequency of updates that are already stable at the receiving replica, a, prior to being propagated to it. We designate such a situation as *pre-reception agreement*. In particular, let u be one such update; i.e. $u \notin sch_a$ and $id_u \in m\text{-}sch_r$. Since u already has its final order with sch_a agreed, the following optimizations may take place before propagating u from some other replica to a by anti-entropy:

1. If $\overline{id_u} \in m\text{-}sch_r$, u needs not be propagated;
2. If $\underline{id_u} \in m\text{-}sch_r$, $precond_u$ may be left empty, as it is no longer needed (in t_3 of Figure 5.2, this is the case of update u before being propagated by anti-entropyfrom A to B);
3. Redundant or self-cancelling updates in a batch of consecutive (according to $m\text{-}sch_r$) stable updates need not be propagated.

5.4 Version Vector Meta-Schedules

A particularly efficient meta-schedule representation is a version vector [PPR+83]. This representation is applicable in systems with the following two properties. First, commitment respects the happens-before relation [Lam78] in the sense that, if update u_1 happens-before update u_2, then the inclusion of $\underline{u_2}$ in a schedule sch_a implies that $\underline{u_1}$ also appears in sch_a, in a previous position than $\underline{u_2}$. Second, propagation of updates ensures the prefix property [PST+97]. Recall from Section 3.2.6 that the prefix property means that, if some replica a holds an update u_i that was issued at a given replica b, then a will also have received all updates that $repB$ has issued before u_i.

For example, Bayou [PST+97], Coda [KS91], and VVWV [BF05b] satisfy the above properties. Furthermore, our baseline solution from Section 3.2.6 also satisfies both properties.

Let each update u in sch_a be time-stamped by a version vector, vv_u, that expresses the happens-before relation between updates.

One may show that, when the above properties hold, the partial order defined by the version vectors of each update is consistent with the total order of any r-schedule in the system.[3] This means that, when the above properties hold, a schedule s, such that $\underline{s} = \underline{u_1};..;\underline{u_n}$, may be identified by a version vector, vv_s, that is the component-wise maximum of each $vv_{u_1},..,vv_{u_n}$.

[3] We use the term consistent in an analogy of Schwarz's and Mattern's notion of consistency with causality [SM94]. In our case, we are not interested in the causality order, but on r-schedule order, however.

Provided that, for all $u \in s$, vv_u is available, vv_s represents a meta-schedule. In fact, from vv_s one can tell that whether, for a given update u, (a) $\underline{id_u}$ is in the meta-schedule (when $vv_u \geq vv_s$); or (b) $\overline{id_u}$ is in the meta-schedule (when vv_u and vv_s are concurrent); or (c) id_u is not in the meta-schedule (otherwise). Further, the order of the update identifiers within the meta-schedule is any that respects the partial order defined by the happens-before relation.

However, if a replica a receives vv_s from a remote replica, a will not always hold vv_{u_i} for every update u_i in the meta-schedule; instead, a has only vv_{u_i} for each update $u_i \in sch_a$. Hence, vv_s may only identify, at each moment, meta-schedules corresponding to updates currently in sch_a. However, such partial representation is sufficient for the Algorithms presented in Section 5.2.1, since they are not concerned with updates identified in meta-schedules until the updates are in sch_a.

5.5 Decoupled Version Vector Weighted Voting Protocol

In this section, we exemplify our decoupled update commitment approach by applying it to the VVWV protocol. The result is an extended variant of VVWV that can progress by exchange of consistency packets.

The VVWV protocol closely fits into the structure of the base protocol abstraction upon which we propose our generic extension for decoupled update commitment. Recalling from Section 4.2, we present VVWV with relation to the *proposeSch* and *getNewDecidedSch* functions from the abstract update commitment protocol from Section 3.2.5. Therefore, having identified such key functions, the decoupled variant of VVWV results from a almost direct application of the generic extension that we describe in Section 5.2.1.

In VVWV, version vectors may be used as meta-schedules, as the protocol satisfies the conditions stated in Section 5.4. In particular, the protocol respects the happens-before relation and the prefix property. Hence, one extends VVWV by applying the abstract extension described in Section 5.2.1 and using version vectors as meta-schedules. Here, the meta-agreement algorithm is the same as described for VVWV, except for the fact that updates are no longer propagated, only version vectors and currency values.

The consistency packets that some replica a issues will hold the $decidedForCommit_r$ and $votes_a[1..N]$ version vectors, as well as the $cur_a[1..N]$ values. If currencies and version vector entries are implemented as integers, then a consistency packet will carry a maximum of $(N+1)N$ integers; this is independent of update activity and update size.

The only aspect that requires special care is candidate proposal in VVWV, as handled by *proposeSch*. More precisely, our decoupled update commitment extension requires modular independence between the *proposeSch* and *constructSch* components. However, such an independence is not present in the original VVWV specification (see Section 4.2), which assumes that the current tentative schedule *always* corresponds to the current proposal voted for by the replica, $votes_a[a]$. Nev-

ertheless, to make *proposeSch* and *constructSch* independent, one must allow the replica to be voting for some proposal that does not coincide with its current tentative schedule.

Hence, we need to change the original *proposeSch* in VVWV to the following new version, denoted $proposeSch - ext$:

Algorithm 21 $proposeSch - ext(schedule\ s)$ at replica a, in extended-VVWV

1: Let *newTentV* be the version vector that identifies the new schedule *s*.
2: **if** $votes_a[r] = \perp$ **then**
3: $votes_a[r] \leftarrow newTentV$;
4: $cur_a[r] = currency_a$.
5: **else if** $newTentV > vote_a[a]$ **then**
6: $votes_a[r] \leftarrow newTentV$.
7: **else**
8: Do nothing.

In contrast to the original *proposeSch* of VVWV, $proposeSch - ext$ checks whether the replica that is issuing the new update has already voted for a candidate that the new schedule *s* does not dominate; in such a case, the local replica cannot vote for its new r-schedule (line 8). Such a situation can occur if the issuing replica has already voted for some concurrent candidate for which it has received the identifier (the corresponding version vector) by consistency packet exchange, but not the corresponding updates.

5.6 Consistency Packet Management

Resilience to malicious non-replicas is achieved by having replicas encrypt and digitally sign consistency packets using a secret *object key*. The object key is associated with the corresponding logical object, and is exclusively known by its rightful replicas. This ensures the privacy and integrity of consistency packets.

The object key also securely identifies the logical object of a consistency packet, as follows. Each consistency packet includes a secure hash of the corresponding object key. A replica may verify that a consistency packet corresponds to a logical object of interest by hashing the object's key that is owned by the replica and comparing the result with the hash contained in the consistency packet. Since such a hash is included in the digitally signed contents of the consistency packet, the integrity of the consistency packet identification is also ensured.

A second issue is buffer management at non-replicas. Non-replicas have limited buffers, which means that, due to contention of consistency packets (from different logical objects and even from the same object), some may have to be discarded. Two techniques may reduce such an occurrence. Firstly, an appropriate erasure method such as those studied in [SHPF04] may avoid obsolete consistency packets consuming network bandwidth and buffer space.

Secondly, a substitution method may prevent older consistency packets from being stored and propagated simultaneously with more recent ones belonging to the same object. A solution is to assign each consistency packet p a version vector $vpacket_p$ with N entries, one for each replica. Accordingly, each replica a also maintains the version vector of the most recent consistency packet it has received or generated, $vpacket_r$. Upon reception of a consistency packet p, replica a merges $vpacket_r$ with $vpacket_p$ ($vpacket_r[k] \leftarrow max(vpacket_p[k], vpacket_r[k])$, for $k = 1..N$); also, upon generation of a new consistency packet p', a increments $vpacket_r[r]$ by one and assigns $vpacket_{p'} \leftarrow vpacket_r$. Finally, each non-replica can compare the version vectors of consistency packets stored at its buffer that pertain to the same object; if $vpacket_{p_1} < vpacket_{p_2}$, packet p_1 may be discarded because all its relevant information is already reflected in the more recent consistency packet p_2.

5.7 Summary

We formally present a generic extension of optimistic replication protocols that decouples the agreement phase from the remaining phases of update commitment. Hence, it enables consistency to evolve, to some extent, by exchange of meta-data that is a small subset of the information exchanged in regular æinteractions. Consequently, potentially memory-constrained non-replicas surrounding the system of mobile replicas may be exploited as carriers of such meta-data.

Non-replicas constitute a low-bandwidth transport channel. Due to decoupled update commitment, such a channel may be used for dissemination of consistency packets, whereas it is prohibitive for anti-entropy. Similarly, our contribution is also applicable to other instances of transport channels with limited bandwidth or high monetary or energy costs, for example.

Chapter 6

Versioning-Based Deduplication

Many interesting and useful systems require transferring large sets of data across a network. Examples include network file systems, content delivery networks, software distribution mirroring systems, distributed backup systems, cooperative groupware systems, and many other state-based replicated systems as the ones that the present thesis targets. Unfortunately, bandwidth remains a scarce resource for most networks [ZPS00, DCGN03], including the Internet and mobile networks. Furthermore, in some cases, network access is an expensive resource, for instance in terms of battery or monetary cost.

As a practical consequence, users are often forced to choose between either using a system that does not perform acceptably, possibly at a significant cost, or simply not using the system and wait until high-bandwidth and inexpensive connectivity become available. Evidently, neither one is a good decision. We need mechanisms for efficient replica synchronization transference across the network.

Although this problem is not new, its importance grows as the above constrained networks become omnipresent. Much recent work has proposed data deduplication techniques, which may be combined with conventional techniques such as data compression or caching. Data deduplication exploits the content similarities that exist across objects (such as files) and across versions[1] of the same object. It works by avoiding the transference of redundant chunks of data; i.e. data portions of the contents to send that already exist at the receiving site. The receiving site may hence obtain the redundant chunks locally instead of downloading them from the network, and only the remaining, literal chunks need to be transferred.

The key challenge of data deduplication is in detecting which data is redundant across the contents to send and the contents the receiver site holds. Most recent

[1] For coherence with the terminology that is common in literature on data deduplication, in this chapter we reason about *versions* instead of *updates*, and implicitly assume that all updates are state-based and whole-object updates. For the same reason, our presentation in this chapter diverges slightly from the definitions in Section 3.2 of Chapter 3. Namely, we consider that the term *version* has the same meaning as a state-based, whole-object *update*; when, according to the definition in Section 3.2, a version only corresponds to the value component of the latter.

solutions rely on the *compare-by-hash* method[2] [MCM01, CN02, BF04, JDT05, ELW+07, BJD06]. Compare-by-hash identifies chunks by a cryptographic hash of their contents; such an identifier is assumed unique due to the low probability of hash collisions. By simple exchange and comparison of chunk hashes, the protocol can then determine which chunks are redundant. Compare-by-hash is very powerful, as it is able to detect any cross-object and cross-version chunk redundancy while maintaining no shared state between the communicating sites.

Nevertheless, compare-by-hash suffers from two major disadvantages. Both result from the intrinsic difficulty of distributed similarity detection. First of all, and contrary to much literature that advocates compare-by-hash, further analysis [Hen03, HH05] argues that this technique is not as risk-free and advantageous as the former often assumes. Despite the low probability,[3] hash collisions may occur, either accidentally or by malicious attack of the hash function [Sch95]; resulting in silent, deterministic, hard-to-fix data corruption [Hen03]. It is very questionable that this possibility is acceptable for most applications and systems.

Second, compare-by-hash adds a significant overhead to the data transfer protocol, in terms of the number of network round-trips, exchanged meta-data volume and local processing time. In situations of low or no chunk redundancy, such an overhead may not compensate for the gains in transferred data volume, and may actually degrade network performance. Moreover, proposed improvements to compare-by-hash [JDT05, ELW+07] to leverage its precision in detecting redundancy come at the cost of increasing at least two of the previous items (see Section 2.7.4.3).

We propose an alternative to compare-by-hash that eliminates the its crucial limitations. It is based on two practical observations.

Firstly, many existing replicated systems store, at each site, a substantial part of the recent version history of each replica. In contrast to bandwidth, storage is a plentiful resource, as disks become cheaper and larger. This makes it possible to log, along with the current versions of replicas, considerable portions of their version histories. In fact, we observe that several storage systems already keep long-term logs, for purposes of recovery from user mistakes or system corruption [SFH+99], backup [CN02, QD02, SGMV08], post-intrusion diagnosis [SGS+00], auditing for compliance with electronic records legislation, such as U.S. Congress Acts HIPAA [Con96] or SOX [Con02] [PBAB07], or versioning support for collaborative work [C+93]. The base solution that lays the ground for the present thesis is an example of one such system.

A second observation is that data similarity in replicated systems mostly propagates across versions under control of the replicated system, in a causal flux of data from earlier versions copied to newer versions of possibly distinct files. Let us designate such similarity relations as *in-band redundancy* (as opposed to out-of-band

[2] There are other usages for this method. Hereafter we use the expression *compare-by-hash* to mean *data deduplication through compare-by-hash.*

[3] Different authors define contradicting expressions for such a probability. Namely, Muthitacharoen et al. [MCM01], Cox et al. [CN02] and Henson and Henderson [HH05].

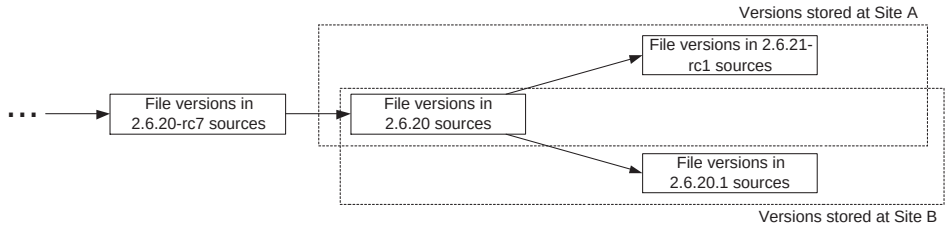


Figure 6.1: Example of replicated system for code development of the Linux kernel sources.

redundancy, which we address later on).

For instance, consider the example of the evolution of Linux kernel source code version that Figure 6.1 depicts. Two sites, *A* and *B*, initially hold a common version, 2.6.20, of the kernel (comprising a directory tree of source code files). Concurrently, the users at each site start working on divergent versions of the kernel. The user at *A* adds new features to the files in version 2.6.20, producing a first release candidate version, 2.6.21-rc1. Meanwhile, the user at *B* fixes bugs and adds security updates to version 2.6.20, producing version 2.6.20.1. Most redundancy that exists between the contents of Linux kernel source code versions 2.6.20.1 and 2.6.21-rc1, which diverge from the common ancestor version 2.6.20, is in-band.[4] It is due to the fact that the file versions of the latter kernel versions mostly result from a series of partial modifications to contents copied from the files in version 2.6.20 of the kernel, which in turn derive from earlier versions, and so on.

Of course, it is easy to devise examples of situations causing out-of-band redundancy. For instance, when two distinct users import identical data chunks from some external data source (which is not under the control of the replication system, such as a web page) and write it on different versions. Nevertheless, as we show in Section 8.3.4.1, empirical evidence suggests that the volume of out-of-band redundancy in the collaborative scenarios that the present thesis addresses is practically negligible, when compared to in-band redundancy.

Essentially, we propose a novel scheme that combines versioning information and local similarity detection algorithms to exploit in-band redundancy. It does not resort to distributed hash comparisons, consequently it does not exploit out-of-band redundancy.

Intuitively, scheme works in four steps. First, the receiver site, *B*, informs the sender site, *A*, of the version set that *B* currently stores (for any replica in *B*), using some space-efficient representation of such set information. Second, the sender site compares *B*'s version set with its own, determining the intersection containing the versions whose contents both sites store in common. Third, using some local similarity detection algorithm, *A* determines which chunks among the versions to send to *B* are redundant with relation to the previous intersection. Any chunk that

[4] Assuming that the contents of all three kernel versions are stored on the same replicated system, hence traceable by the latter.

A finds redundant with the common version sets is guaranteed to be redundant with relation to the contents that *B* currently stores. Finally, *A* transfers the contents of the remaining, literal chunks of the versions to propagate, and replaces the contents of the redundant chunks by a reference to one common version where *B* can locally retrieve them.

Let us illustrate the above steps by recalling the previous kernel development example. Assume that, after both sites produce divergent versions of the contents of the initial kernel version (2.6.20), site *B* initiates a synchronization session from *A*. Site *B* sends sufficient information to *A* to let it know that *B* currently stores the file versions of the 2.6.20 and 2.6.20.1 kernel versions. From such information, *A* will be able to determine that both sites hold the contents of kernel version 2.6.20 in common. Then, before sending the new versions that *A* holds (those in kernel version 2.6.21-rc1), *A* checks whether such new versions share any chunks with the contents of kernel 2.6.20; since all such versions are available locally, this step is local. At the end, *A* needs not send the contents of the chunks that it found redundant, replacing them by a simple reference that tells *B* where, within the versions *B* stores for kernel 2.6.20, *B* may obtain the redundant contents.

Overall, our solution achieves three fundamental properties. First, the actual process of similarity detection is local, unilaterally performed at the sender site across the data it stores (third step). Consequently, such a step is no longer constrained by network bandwidth, as happens with compare-by-hash solutions. We may therefore employ more data-intensive techniques, which both avoid the possibility of false positives and can detect more in-band redundancy. Furthermore, shifting similarity detection to the local context ensures a very small network overhead and no additional network round-trips.

Second, the whole method may be achieved by deterministic techniques; again in contrast to compare-by-hash, which is probabilistic. The first step is a well studied problem in the domain of version tracking in replicated systems [RS96], which we solve very efficiently using Vector Sets [MNP07], a variant of version vectors. The second step, as mentioned above, is now able to detect chunk redundancy deterministically.

Finally, provided that:

1. out-of-band redundancy occurs at sufficiently low levels, when compared to in-band redundancy, and

2. the logs that each site holds cover a sufficiently deep version history of its replicas,

then the redundant volume that our approach can detect is comparable to what any compare-by-hash solution may potentially achieve (i.e. both in-band and out-of-band). On one hand, the first condition implies that most redundancy will be in-band. On the other hand, the second condition means that our approach will be able to detect most of such dominant in-band redundancy. The reason to that being that, under condition 2, the intersection between the version sets from both

synchronizing sites will often be large enough to connect transitive relations of in-band chunk redundancy from the contents to send to the contents at the receiver site.

Summing up, our contribution may be seen as a pragmatic alternative to compare-by-hash. It ignores out-of-band redundancy, trading it for safer and more efficient deduplication of in-band redundancy. Overall, the outcome may be considerably advantageous in most systems and workloads, a claim we support with experimental results in Section 8.3.

As a secondary contribution, we propose novel compression schemes for local log storage. They take advantage of redundancy across local versions, obviating the actual storage of redundant chunks. Evidently, such compression schemes contribute to achieve the above second condition, as logs take up considerably less space, naturally delaying the need to prune old versions from logs.

We describe our approach in the scope of a distributed archival file system, called dedupFS. Yet, our approach may be generically applied to other kinds of systems. In particular, we may directly apply it to improve the initial solution that we have introduced in Section 3.2.6.

In the following, we start by laying out the basic data structures that comprise the state of each dedupFS site, in Section 6.1. We then present the data deduplication protocol in Section 6.2, which relies on such a state. Section 6.3 describes how long-term version logs are efficiently stored and kept within the limited storage capacity of each site. Section 6.4 then addresses the maintenance of chunk redundancy relations among the local version contents that a site stores. Section 6.5 discusses the problem of out-of-band redundancy. Finally, Section 6.6 summarizes the chapter.

6.1 Simple Transfer Synchronization Protocol

dedupFS provides replicated file system services to a number of distributed sites. Essentially, dedupFS replicates, at each site, a number of files and directories. It offers local access to replicated files and directories through a standard file system API. dedupFS is an archival file system, offering access to a partial history of the versions of the replicated objects.

We are mainly concerned with file synchronization efficiency, which should dominate storage and network usage in dedupFS.[5] So, for simplicity of presentation, this chapter intentionally excludes directory hierarchies and considers a flat file space; as Section 7.2 describes, the actual implementation of dedupFS fully supports multi-level directory trees.

dedupFS divides the logical file space into disjoint sets of files, called *r-units*. The minimum replication grain in dedupFS is the individual r-unit. Each site may

[5] Efficient directory storage and synchronization has already been addressed elsewhere (e.g. [KS93],[SGSG02]).

replicate any arbitrary set of r-units. However, a full r-unit is completely replicated, i.e. a site replicating a r-unit maintains replicas for all files of that r-unit.

As will become evident in the remainder of the section, larger r-units allow the synchronization protocol to exchange less meta-data and perform faster local deduplication processing; while smaller r-units enable finer granularity in the choice of which versions to prune and which files to replicate. A particular case is the initial solution from Section 3.2.6, in which each r-unit is atomic, i.e. it corresponds to one object. Clearly, the meta-data overhead of atomic r-units (e.g., in the initial solution, due to one version vector per replica) makes such an option hardly scalable to large numbers of replicas.

Given a set of files, their optimal partitioning in r-units is out of the scope of the thesis.[6] Hereafter, we assume that users and applications explicitly construct r-units by grouping related files together; for instance, the working-set of a collaborative project, such as the files of the Linux kernel project, can form a r-unit.

For each r-unit, there is a set of sites that replicate the r-unit. Such sites may create new files in the r-unit, as well as read and modify existing files in the r-unit. For simplicity of presentation, and without loss of generality, the set of sites replicating each r-unit is static and each of its members has a unique identifier.

6.1.1 Versioning State and Synchronization

When two sites replicating a common r-unit synchronize, they need to determine the minimum set of file versions from the r-unit that the sender site needs to transmit to bring the receiver site up-to-date. They do that by exchanging an efficient representation of the set of versions each site stores for that r-unit.

We use a representation of version sets that derives from Vector Sets (VSs) [MNP07] (which Section 2.5.6 analyzes). VSs represent the set of versions of a given r-unit by a single vector, with one counter per site replicating the r-unit, provided that every synchronization session completes without faults [MNP07]. In contrast to the solution of one version vector per object in the r-unit, VSs scale well to large numbers of objects, provided that the number of r-units is kept low and that faults during synchronization are rare.

VSs are originally designed for state-based systems that do not log replica versions, thus simply store the most recent version each replica knows of.[7] In order to fit VSs with our log-based architecture, we need to adapt VSs and the corresponding synchronization protocol to our log-based model.

We now partially redefine our initial solution in order to incorporate r-units and our variant of VSs. Since we are only concerned with synchronization, we do not address update commitment in this chapter; Section 6.1.3 discusses how we can

[6] A possible partitioning strategy may be to select root directories holding the files concerning a given task, belonging to a given team of users, as r-units.

[7] Therefore, the original VS synchronization protocol maintains only the knowledge vector. Since it does not consider version logging, the pruned knowledge vector, which we introduce next, is not necessary in the original VS solution.

complement the solution we propose next with the update commitment protocols from the previous chapters.

Similarly to the initial solution, we store, along with each file version, a version identifier consisting of a pair $\langle sid, lid \rangle$. Such a pair is system-wide unique. When a site replicating the r-unit creates a new version, it identifies it by the site identifier and by a locally-unique, monotonically-increasing number. For example, the versions created by site A within a given r-unit have identifiers $\langle A,1 \rangle$, $\langle A,2 \rangle$; and so on. Due to faulty synchronization, a version may also be temporarily assigned a predecessor vector, as defined in [MNP07]. For presentation simplicity, and without loss of generality, we assume that faults never occur during synchronization and thus omit predecessor vectors from our description.

Any site, A, maintains, along with each replicated r-unit u, two vectors: a *knowledge vector*, denoted $KV_A(u)$, and a *pruned knowledge vector*, denoted $PrV_A(u)$. $KV_A(u)$ identifies the set of versions (in u) that happen-before the versions that the site currently stores in r-unit u (including the latter versions). $PrV_A(u)$ identifies the subset of the set denoted by $KV_A(u)$ that was pruned from the log and is no longer available at the site. The set of versions that A currently stores for r-unit u is given by the set difference beween the set denoted by $KV_A(u)$ and the set denoted by $PrV_A(u)$.

Both vectors start out null. Naturally, $KV_A(u)$ changes as A creates or receives new versions at u, while $PrV_A(u)$ changes as A prunes versions at u. As site A adds a new version of a file, $\langle B,c \rangle$, to a r-unit u, the entry in $KV_A(u)$ that corresponds to site B (denoted $KV_A(u)[B]$) is updated to c, if c is higher than the previous entry. Vector $PrV_A(u)$ is updated accordingly for each version that A prunes in u.

Given a version v, identified by $\langle B,c \rangle$, of a file of a given r-unit u, then v is in the knowledge (resp. pruned knowledge) set of u at site A, iff $c \leq KV_A(u)[B]$ (resp. $c \leq PrV_A(u)[B]$).

We may finally redefine the base synchronization protocol that we introduced for the initial solution in Section 3.2.6, in order to use r-units and knowledge vectors. We present it in Algorithm 22.

A synchronization session between sites A and B starts by having B send the identifiers of the r-units it replicates, as well as its knowledge vectors (line 2). Accordingly, A determines the subset of the previous r-units that are common to both sites (line 3), and proceeds to send the new versions of each common r-unit to B.

Before synchronizing each common r-unit, A needs to ensure that, by sending the new versions that A holds to B, B will still ensure the prefix-property (see Section 3.2.6); such a property is required for correctness of our deduplication scheme. However, if A has already pruned at least one version of some object in a given common r-unit and B has not yet received such a version, then if A propagates more recent versions of the same object to B, there can occur a gap at B's logs that violates the prefix-property. To avoid such a situation, we require that the pruned knowledge of A be a subset of the knowledge of B (line 5). Otherwise, we simply forbid synchronization of this r-unit between this pair of sites.

Algorithm 22 Update propagation protocol from site A to site B.

```
 1: for all r-unit u that B replicates do
 2:     B sends the identifier of u, along with KV_B(u) and PrV_B(u), to A.
 3: A determines the set of r-units (denoted R) that both A and B replicate.
 4: for all r-unit u in R do
 5:     if PrV_A(u) ≤ KV_B(u) then
 6:         for all Object x that A replicates (with replica a) and belongs to u do
 7:             A sends the identifier of x to B.
 8:             B looks up a local replica of x, denoted b.
 9:             if B does not replicate x then
10:                 B creates an empty replica of x, denoted b.
11:             Let u be the first update in sch_a.
12:             while u exists do
13:                 if KV_B(u)[u.rid] < u.lid then
14:                     A sends u to B.
15:                     B sets sch_b ← scheduleUpdate(u, sch_b).
16:                 Let u be the next update in sch_a.
17:     else
18:         Missing versions among both sites; skip synchronization of r-unit u.
19: for all u ∈ R synchronized successfully do
20:     A sends KV_A(u) to B.
21:     B merges the received KV_A(u) with KV_B(u) into KV_B(u).
```

In the latter case, B will only be able to synchronize the previous r-unit with some other site that has not yet pruned the necessary versions. Evidently, it is crucial that one such site does exist. In practice, we may consider that there exists some back-up site, with large storage resources, whose purpose is to store sufficiently long logs, in order solve any situation as above that may arise.

In contrast, if the prefix-property is guaranteed between A and B for some common r-unit, A propagates the versions (of the objects in the r-unit) that A holds but B does not (by checking the condition at line 13). For each version B receives, B schedules it in the log of the corresponding replica. If such a replica does not yet exist at B (since B is not yet aware that some site created the corresponding object into the r-unit), B creates an empty replica of the new object (recall that r-units are fully replicated), at line 10.

Finally, B updates the knowledge vectors of the r-units for which synchronization was carried out, by merging B's each knowledge vector with A's (line 21).

With regard to the above protocol, we are concerned with the content transfer at line 14. Furthermore, we also need to ensure that each replica's version log, as needed by the above protocol, is stored efficiently.

Before describing our contribution to optimize such aspects through data deduplication, we draw some considerations on log pruning and update commitment with the above protocol.

6.1.2 Log Pruning and Pruned Knowledge Vectors

Similarly to the initial solution, sites prune versions from replica logs according to the corresponding log-order, starting at older versions. This ensures that pruning does not violate the prefix-property, on which our deduplication scheme relies.

Pruning, however, may cause the pruned knowledge vector to represent a set that is greater than the effective set of pruned versions. We illustrate with a simple example. Consider two objects, x and y, belonging to the same r-unit, u. Let us assume that some site, A, replicating the r-unit, stores, at the log of x's replica, version $\langle B,1\rangle$; while, at y's replica, the site logs versions $\langle B,2\rangle$.

Assume that the pruned knowledge vector was, at this moment, a null vector. If the site decides to prune the oldest version from y's log ($\langle B,2\rangle$), it will update the respective pruned knowledge vector with such a version. Accordingly, $PrV_A(u)[B]$ will become 2. This means that the pruned knowledge set will now represent a set that contains both versions $\langle B,1\rangle$ and $\langle B,2\rangle$, while in fact the site has only pruned the former.

The first immediate consequence of such an imprecision is that the condition at line 5 of Algorithm 22, which analyzes the pruned knowledge vector of the sender site, may yield false negatives. Therefore, the previous algorithm may prohibit synchronization of some r-units between a given pair of sites because it assumes that the prefix-property will be violated when, in fact, it would be safe. Nevertheless, it is not a critical limitation, as it does not violate synchronization safety.

We believe this is an inevitable consequence of using a pruned knowledge vector that covers a whole r-unit, while pruning occurs at the finer granularity of individual objects within the r-unit. Two alternative solutions may eliminate the above false negatives, introducing different drawbacks. A first solution is to use smaller r-units. The smaller the r-unit, the lower the probability of situations as above (the limit being atomic r-units, where such a situation is impossible); however, smaller r-units imply higher overhead with knowledge vectors, due to more r-units to represent the same amount of objects. A second solution is to further restrict the pruning order to respect the order at which a given site created versions within the r-unit; i.e., in the above example, the first version to prune would necessarily be $\langle B,1\rangle$, which belongs to object x. This solution has the important drawback of stealing the choice of which object from which to prune from the user, imposing a system-defined order.

6.1.3 Update Commitment and dedupFS

dedupFS, by itself, does not include any update commitment protocol. It may, however, be combined with the commitment protocols that we propose in Chapter 4, hence producing a more complete replicated system.

dedupFS imposes one important condition to the commitment protocol that may complement it: that any update that the commitment protocol decides as aborted be still propagated to the replicas that do not hold the corresponding ver-

sion. In the VVWV protocol (see Section 4.2), for instance, we can optimize network usage by not propagating updates that have already been decided as aborted. If VVWV is to be used as a complement of dedupFS, such an optimization must be turned off. Nevertheless, as Sections 8.1 and 8.3 show, experimental results show that, under realistic update ratios, aborted updates are rare; while, in real workloads, the bandwidth gains of dedupFS largely surpass the penalty of having to transfer a small number of aborted updates.

With respect to the VVWV protocol (see Section 4.2), combining it with dedupFS has the problem that the former relies on per-replica version vectors (one per replica), while the latter on per-r-unit knowledge vectors.

One direct solution is to let both representations co-exist in the integrated system. Nevertheless, this solution increases the space overhead with version tracking, which is redundantly done at the layers of update propagation (dedupFS) and update commitment (VVWV).

A second solution is to use atomic r-units, in which case knowledge vectors effectively consist of the same conventional version vectors that VVWV uses. With this solution, both dedupFS and VVWV could share the same version tracking representation. However, atomic r-units imply important performance and efficiency penalties for dedupFS, when the number of files (thus, atomic r-units) is large.

In our opinion, neither solution is entirely satisfactory. A direction for future work is, inevitably, to adapt VVWV to non-atomic r-units and knowledge vectors.

6.2 Distributed Data Deduplication Protocol

Having described the basic data structures and the synchronization protocol for plain transfer, we now extend dedupFS so as to achieve distributed data deduplication during synchronization.

6.2.1 Chunk Redundancy State

Each site maintains state about the content redundancy across its replicas. Such a state connects identical disjoint, contiguous portions of the set of versions that the site stores. We call each such portion a *chunk*. Furthermore, given a set of versions, if two chunks taken from a set of versions have identical contents, we designate them as redundant across the set of versions. Chunk redundancy may either occur between chunks of the same version, versions of the same file, distinct files, or even distinct r-units. If otherwise, a chunk has no other chunk, within the version set, with identical contents, we designate the former as literal.

Conceptually, we may represent chunk redundancy relations among the versions that a site stores as a set of two-level *similarity trees*. Given a set of chunks sharing identical contents, we distinguish one chunk as the *root chunk*, and the remaining chunks as the *child chunks*. While the root chunk is guaranteed to always have its contents available in-place, the system may discard the (redundant)

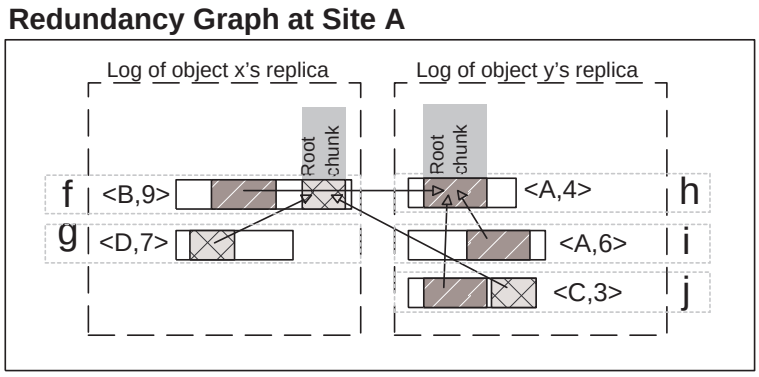


Figure 6.2: Example of chunk redundancy graph. Site *A* replicates objects *x* and *y*. For each replica, *A* currently logs the above versions (denoted f, g, h, i, j), identified by their unique version identifiers. The five versions share identical chunks. A possible similarity graph that conceptually represents such redundancy is depicted. Two chunks are chosen as root chunks, and the remaining chunks with identical contents are their child chunks.

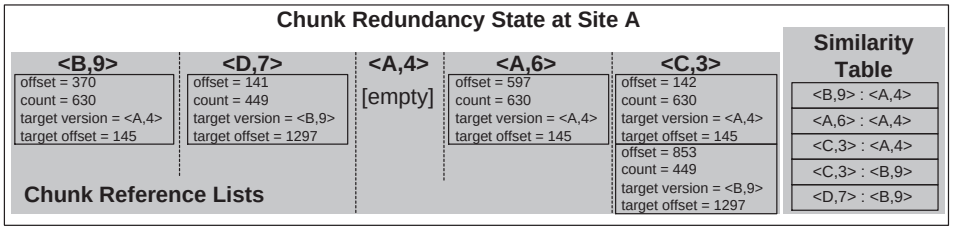


Figure 6.3: A possible chunk redundancy state is depicted. Example of chunk redundancy state. Site *A* stores the versions of Figure 6.2 (f, g, h, i, j), identified by their unique version identifiers ($\langle B, 9\rangle, \langle D, 7\rangle$, etc). Site *A* internally represents such a state by the chunk reference lists and the similarity table above.

contents of child chunks for storage efficiency (as Section 6.3 explains). As an example, Figure 6.2 depicts a possible set of similarity trees for a set of versions, f, g, h, i and j.

In practice, a site internally represents its redundancy graph by means of two data structures, which form its chunk redundancy state. Figure 6.3 depicts such a state for a site holding the redundancy graph of the previous example. The first data structure consists of a chunk reference list associated with each version the site stores. The chunk reference list of a given version identifies the chunks of the version that are children in some similarity tree. Each chunk reference identifies: the child chunk (by its offset and length within the version the reference belongs to) and the corresponding root chunk (by the identifier of the version where the root chunk is located and the corresponding offset within that version).

The chunk reference lists of all versions in a site are sufficient to construct its redundancy graph. However, for performance, each site also maintains a *similarity table*. The similarity table stores pairs of $\langle$ child version identifier, root version

identifier $\rangle$; where each pair means that the version identified by the child version identifier has at least one chunk that is a child of a root chunk in the value identified by the root version identifier. The similarity table accelerates the detection of chunk redundancy during synchronization, as will become clear in the next section.

For the moment, we do not explain how a site maintains its chunk redundancy state as its replicas contents evolve. Section 6.4 addresses such an issue.

6.2.2 Data Deduplication Protocol

Having described the state that each site maintains, both concerning the set of locally stored versions and the chunk redundancy across such a set, we now explain how we exploit such a state for data deduplication during synchronization. Let us consider a synchronization session from a site *A* to a site *B*, as defined in Algorithm 22 (see Section 6.1.1). In particular, let us focus on the line where actual version transference occurs (line 14).

We replace such a line with the following three steps, which ensure distributed data deduplication:

1. *B* sends the $KV_B(u_i)$ and $PrV_B(u_i)$ of each r-unit u_i that *B* replicates. For each r-unit, *A* determines, by comparison of the pruned knowledge and knowledge vectors that it receives from *B* with its own vectors, two sets of versions: (i) $\mathcal{T}$, the minimum set of new versions that *B* does not yet store, and thus should be transferred; and (ii) $\mathcal{C}$, the maximum set of versions that both sites store in common.

2. *A* analyzes its local redundancy state to identify which chunks are redundant between $\mathcal{T}$ and $\mathcal{C}$. Such chunks are guaranteed to be redundant between the versions to transfer and the versions that *B* already holds, thus need not be transferred.

3. Finally, a third step propagates the contents of literal chunks in $\mathcal{T}$, as well as lightweight references to chunks in common versions in the case of redundant chunks.

Figure 6.4 depicts our data deduplication protocol. We detail each step in the following sections.

6.2.2.1 $\mathcal{T}$ and $\mathcal{C}$ Set Determination

By simple comparison of the received vectors with its own vectors, *A* is able to define the membership conditions for the $\mathcal{T}$ and $\mathcal{C}$ sets. Let $\mathcal{R}$ be the set of r-units that both *A* and *B* replicate. Consider a version *v* stored by *A*, identified by $\langle D, c\rangle$, belonging to r-unit *u* in $\mathcal{R}$.

Then $v \in \mathcal{T}$ if and only if:

$$KV_B(u)[D] < c \tag{6.1}$$

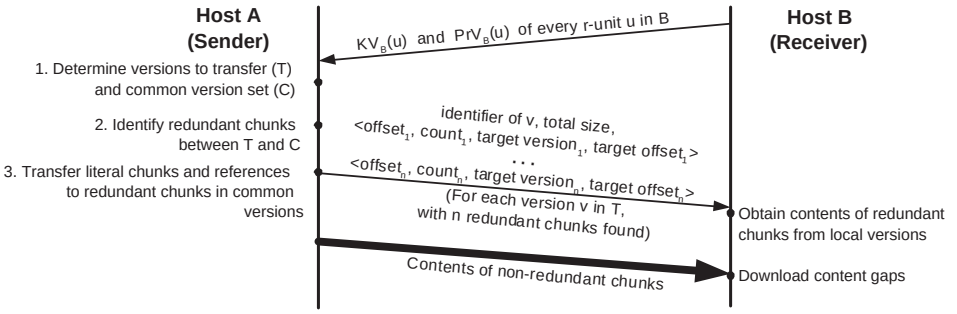


Figure 6.4: Data deduplication protocol.

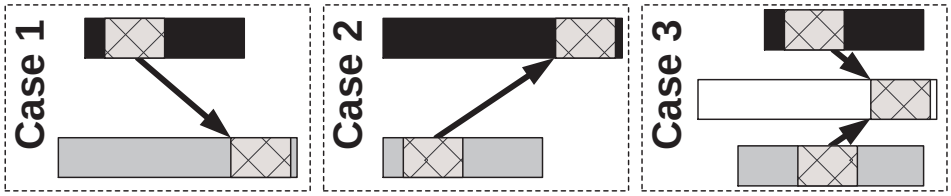


Figure 6.5: The three possible cases of chunk redundancy between versions in $\mathcal{T}$ and $\mathcal{C}$. The version in black is in $\mathcal{T}$, while the gray version belongs to $\mathcal{C}$.

Furthermore, $v \in \mathcal{C}$ if and only if:

$$\exists k \in \mathcal{R} : PrV_B(k)[D] < c \leq KV_B(k)[D] \tag{6.2}$$

Equation 6.1 captures the versions that do not yet belong to the knowledge set of the receiver replica regarding the r-unit u (otherwise, $KV_B(u)[W] \geq c$). Such versions are necessarily new versions that the receiver site has not received before. Equation 6.2 selects the versions that site A stores and, having been received by site B, at some replica of some r-unit k ($c \leq KV_B(k)[D]$), have not been pruned at B ($PrV_B(k)[D] < c$). Therefore, such versions are currently available at both sites.

6.2.2.2 Chunk Redundancy Identification

For each version, v, to transfer (i.e., $v \in \mathcal{T}$), A resorts to its local chunk redundancy state to identify the chunks that are redundant across $\mathcal{T}$ and $\mathcal{C}$. Take each chunk of v. Clearly, such a chunk is redundant if at least one chunk of its similarity tree belongs to a value in $\mathcal{C}$. This may happen in three different cases, which Figure 6.5 depicts. Namely:

- Case 1. A chunk in v has a root chunk in a version in $\mathcal{C}$.
- Case 2. A chunk in v is a root chunk of a child chunk in a version in $\mathcal{C}$.
- Case 3. A chunk in v has a root chunk which has a child chunk in a version in $\mathcal{C}$.

We look for case 1 by iterating over the chunk reference list of v and checking if each referenced version, v_r, is in $\mathcal{C}$, using the membership condition of $\mathcal{C}$. For cases 2 and 3, we need an auxiliary function, `getInboundRefs`, that determines the set of chunk references of versions in $\mathcal{C}$ that point to a given target value. `getInboundRefs` accesses the similarity table to identify versions that have any reference to the target version. Then, it iterates over the chunk reference lists of each such version and returns the chunk references that effectively point to the target version.

By calling `getInboundRefs` for v, we directly detect the chunks of v in case 2. For case 3, we iterate over the chunk reference list of v and, for each referenced root chunk, we call `getInboundRefs` on the latter's version; we then identify which references both (i) are from versions in $\mathcal{C}$ and (ii) point to the offset of the given root chunk.

So far, we have considered that $\mathcal{C}$ contained the versions that both sites hold prior to synchronizing, and have identified chunk redundancy across such versions. We can still widen $\mathcal{C}$ by observing that, since we propagate versions in $\mathcal{T}$ in a total order, when B receives v, B will already have received every version sent before v. Clearly, the latter versions are already common to both sites at the time v is about to be sent; hence, $\mathcal{C}$ should also include them. The algorithm achieves this by keeping a set of identifiers of the versions that it has already processed in a set $\mathcal{C}'$. When testing inclusion in $\mathcal{C}$, it also checks for inclusion in $\mathcal{C}'$. As a result, we are able to identify chunk redundancy also across the values in $\mathcal{T}$. In the remainder of the thesis, we call the above situation *recursive redundancy*.

6.2.2.3 Chunk Transference

Having identified the redundant chunks, we are now able to efficiently transfer the versions in $\mathcal{T}$. For each version, we transfer three components. A first component identifies the version identifier and the corresponding size. The second component is an array of chunk references that identify where the target site, B, may locally obtain each redundant chunk of the version. We send references in the order by which the corresponding chunks appear in the version. Each chunk reference identifies one chunk in $\mathcal{C}$ that has similar contents. For each case (1, 2 and 3), it is straightforward to determine such a chunk in $\mathcal{C}$. The third component consists of the contents of every literal chunk of the version, again in their order of appearance within the version.

Upon reception of the above components, B may reconstruct the version by copying the referenced contents from the local versions to the specified offsets. The gaps are filled with the contents in the third component.

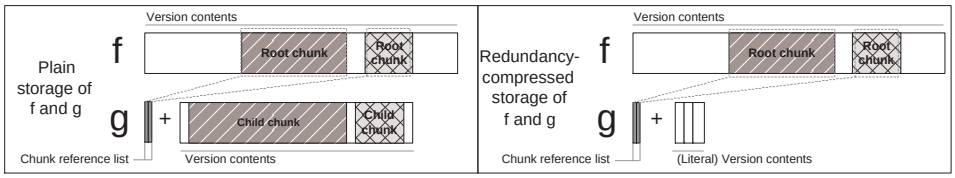


Figure 6.6: Example of redundancy-compression of versions f and g.

6.3 Efficient Log Storage

The effectiveness of the data deduplication protocol of dedupFS depends on the maintenance, at each site, of long-term logs for each replica. Although disk space is normally an amply available resource, a large number of files may easily consume it if logs are not stored efficiently.

This section presents two ways of compressing stored logs, which substantially reduce their space requirements. We designate the first by redundancy-compression and the second by partial log pruning. Both may still be complemented by conventional methods, such as data compression, to further increase space efficiency.

6.3.1 Redundancy-Compression

Redundancy-compression excludes redundant child chunks from the stored representation of a version. A redundancy-compressed version stores only its chunks that are, according to the local chunk redundancy state, literal. We redundancy-compress a version by a simple iteration of the respective chunk reference list, removing the child chunks in the list from the stored contents of the version and coalescing the remaining (literal) contents.

Figure 6.6 depicts redundancy-compression of a site with two versions. Clearly, redundancy-compression is lossless, since all the contents of the version remain available. However, in the case of read accesses to portions of a version that belong to a child chunk (in the example, version g), extra random disk accesses are required to fetch the desired contents from the root chunks that reside at other versions. Hence, the performance cost of redundancy-compression grows with redundancy, as the more redundant chunks in the version, the more child chunks exist.

Being an archival file system, a significant amount of the log of a file will comprise archived versions, and not current versions. Since the former are typically rarely accessed and, when accessed, the performance expectations of applications are not as high as for the latter, such a penalty in access performance is acceptable with archived versions. The same does not apply for most current versions; those should be left uncompressed.

6.3.2 Partial Log Pruning

In cases where the storage savings by redundancy-compression are not sufficient, one step to resort to is to prune old versions from replica logs. However, log pruning, in contrast to redundancy-compression, is a lossy method of reclaiming storage, which erases the literal contents from the pruned versions. Even if the pruned versions were already obsolete from the local user's point of view, discarding shortens log depth. As discussed previously, this can hinder dedupFS's ability to detect distributed in-band redundancy, as version set intersections between two sites may become smaller, possibly breaking the chain of redundancy relations from the versions to send ($\mathcal{T}$) to the versions in common ($\mathcal{C}$).

We may, instead, opt for an intermediate solution, which we call *partial log pruning* (or simply partial pruning). The idea of partial pruning is to discard the literal contents of the versions to prune, which typically dominate the storage requirements of such versions, but still keep their *redundancy footprint*. The redundancy footprint of a version is a lightweight data structure that may partially replace the pruned version for the sake of deduplication. As we describe next, the redundancy footprint allows dedupFS to detect most redundancy that it would detect had the version not been pruned.

The redundancy footprint of a version *v* consists of its chunk reference list. In order to partially prune *v*, we proceed in the following steps:

1. For any root chunk in *v*, we look for one of its child chunks that belongs to another (not yet partially pruned) version. If one such child chunk exists, then we update the chunk redundancy state so that the child chunk becomes the new root chunk (Section 6.4.3 details such an operation).

2. We discard the entire stored contents of *v* (again, updating the chunk redundancy state accordingly, as Section 6.4.3 describes).

3. We mark the log entry corresponding to *v* as partially pruned. The corresponding chunk reference list implicitly becomes *v*'s redundancy footprint.

4. We update the pruned knowledge vector in order to include *v* (as we describe in Section 6.1.1 for full log pruning).

As happens with versions, we cannot store redundancy footprints for ever. Similarly to versions, we may also prune partially pruned entries from replica logs, as long as such a pruning starts by the oldest entries. In order to track which redundancy footprints a site has already discarded, each site *A* maintains a third vector along with each r-unit *u*, which we designate the *untraceable knowledge vector*, or simply $UntrV_A(u)$.

The traceable knowledge vector evolves analogously to the pruned knowledge vector; when site *A* prunes the partially pruned entry of some version identified by $\langle B, c\rangle$ (belonging to some r-unit *u*), it updates $UntrV_A(u)[B]$ to *c* if $c >$

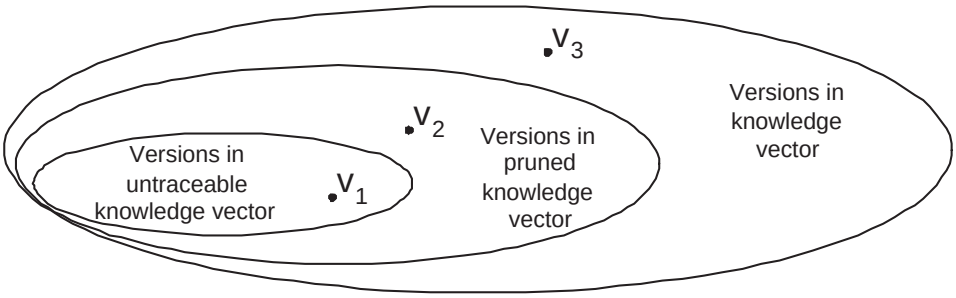


Figure 6.7: Example of the version sets that each knowledge vector represents, for some site. Versions v_1, v_2 and v_3 have already been received and logged at the site. Assuming that no false inclusion has occurred (see Section 6.1.2), versions v_1 and v_2 have already been partially pruned. Version v_1 has been fully pruned.

$UntrV_A(u)[B]$. Similarly to the pruned knowledge vector, the untraceable knowledge vector suffers from false inclusions, as Section 6.1.2 explains.

Summing up, we may now distinguish three stages in the lifecycle of a log entry, each captured by each knowledge vector. Figure 6.7 illustrates the evolution of such three stages. Initially, the entry is created and inserted to the log, hence the corresponding version is added to the knowledge vector of the site. Secondly, the version may be partially pruned, having its literal contents lost and being added to the pruned knowledge vector. Finally, the version may be fully pruned, having its redundancy footprint erased and being added to the untraceable knowledge vector.

dedupFS takes advantage of redundancy footprints for distributed deduplication. As we describe next, we may exploit redundancy footprints exclusively at the sender site or at both sites.

6.3.2.1 Exploiting Sender-Side Footprints

Recall that, originally, dedupFS's deduplication protocol relies on a set intersection between the versions that both synchronizing sets know and have not yet pruned (either partially or fully) (see Section 6.2.2. Redundancy footprints allow dedupFS to enlarge the above condition to encompass more versions; hence achieving a larger interception set and, consequently, improved ability to detect in-band redundancy.

More precisely, since redundancy footprints provide us with some information concerning the partially pruned versions (in contrast to the fully pruned ones), we may also consider them when calculating the intersection. Such additional inclusion has some implications on the protocol, which depend on whether we exclusively consider partially pruned versions at the sender site or at both sites.

We start by the easier case, the first one. Adapting the deduplication protocol from Section 6.2.2 to also consider the partially pruned versions that the sender site holds is straightforward. When identifying redundancy across the new versions to propagate (set $\mathcal{T}$) and the common version set (the intersection set $\mathcal{C}$), we now

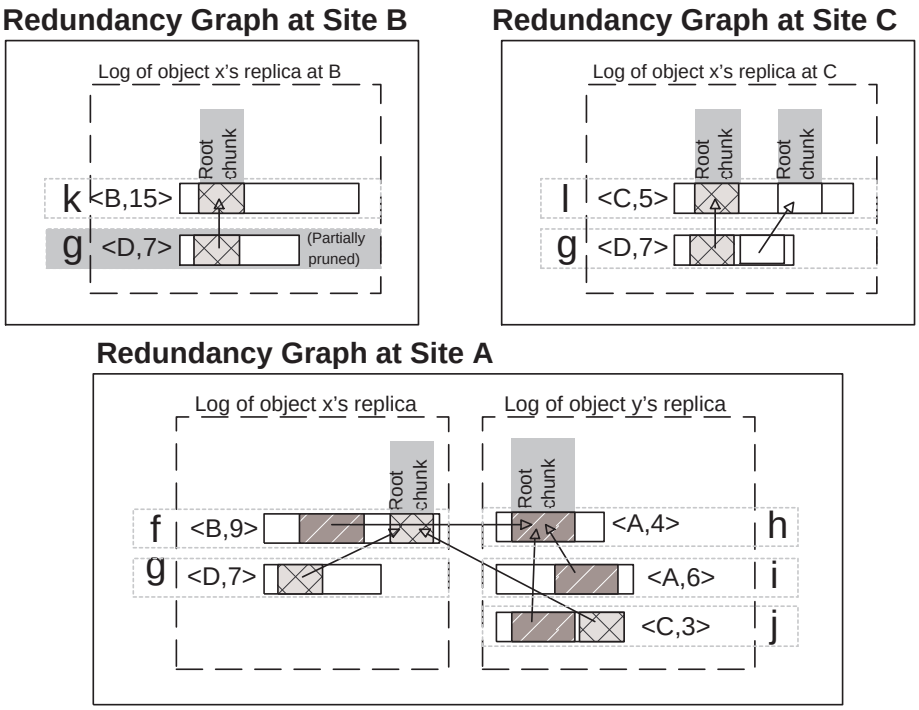


Figure 6.8: Example of the redundancy graphs at of three sites replicating a common r-unit. Site *B* has poor storage resources, and has partially pruned version *g* (identified by $\langle D, 7\rangle$

.

consider both unpruned as well partially pruned local versions as possible members of $\mathcal{C}$ (in the algorithm described in Section 6.2.2.2).

Note that the deduplication protocol exclusively uses chunk references from the local versions in $\mathcal{C}$ that happen to share some chunk with some version in $\mathcal{T}$, and never the contents of the former. Since redundancy footprints of partially pruned versions hold the former information, the above adaptation is sufficient.

We illustrate it with an example taken from the system that Figure 6.8 depicts. Let us consider a synchronization session from site *B* to *A*. After receiving *A*'s knowledge vectors, *B* knows that it should transfer version *k*, as *A* does not hold it yet. If version *g* were fully pruned at *B*, the common set of versions of both sites would be a null intersection; hence, *B* would have to transfer the entire contents of *k*.

Nevertheless, if we enable exploitation of partially pruned versions in the deduplication, then *B* concludes that partially pruned version *g* is common to both sites. From the redundancy footprint of *g*, *B* determines that one chunk of *k* is redundant across both sites. Hence, *B* avoids sending the contents of such a chunk. Instead, it sends a chunk reference to version *g*, including its version identifier ($\langle D, 7\rangle$) and the the offset and size of the chunk. All such elements are available from *g*'s footprint.

6.3.2.2 Exploiting Footprints at the Receiver Side

Let us now try to exploit partial pruned at the versions at the receiver side. A naive adaption of the deduplication protocol to consider the partially pruned versions at the receiver site would simply replace the pruned knowledge vectors that the receiver site sends by the respective untraceable knowledge vectors. More precisely, for each r-unit that a receiver site B replicates, B would send $UntrV_B(u)$ instead of $PrV_B(u)$ (in line 2 of Algorithm 22 from Section 6.1.1).

However, such an adaptation is not always correct, as we now explain.

In contrast to the sender-side case, the deduplication protocol involves obtaining contents of chunks in the local versions that belong to the common version set ($\mathcal{C}$) that the sender determined. If we allow some of such local versions in $\mathcal{C}$ to be partially pruned versions, there will be situations where the receiver site will receive a chunk reference to contents in such partially pruned versions.

Two cases may happen in such a situation: (a) the referenced chunk is redundant in the receiver site, thus it is available at some unpruned version; or (b) the referenced chunk was literal by the time the containing version was partially pruned, which means that the contents of the chunk are no longer locally available. Using the redundancy footprint of the partially pruned version, we can detect in which case we are, for each chunk reference that the site receives. Essentially, if the contents covered by the footprint's chunk reference list include the chunk that the received chunk reference points to, then we are in case (a). In such a case, we may obtain the required contents from the local, unpruned versions that the footprint references.

Otherwise, we are in case (b). This case means that there are some contents that the sender site assumed that would be locally available at the receiver site, yet are no longer locally available. The only solution in this case implies a second protocol round, in which the receiver site explicitly requests a plain transfer (i.e. without deduplication) of the missing chunks.

We illustrate both cases by recalling the example in Figure 6.8. Let us assume synchronization from C to B. Site C has one new version that B has not yet received, l. As the Figure depicts, version l shares two distinct chunks with version g, which is partially pruned at B. If we disable redundancy footprint exploitation at the receiver side, then C will consider that both sites share no version in common (i.e. $\mathcal{C} = \emptyset$). Hence, C will send the entire contents of g to B.

However, if we exploit redundancy footprints at the receiver side, then C will infer that there is, in fact, a common version (g) in relation to which version l has two redundant chunks. Thus, C will not send the contents of such two chunks to C, instead sending two chunk references to chunks in $\langle D, 7\rangle$ (i.e. g). Upon receiving both chunk references, B will analyze g's footprint and infer that: (a) the contents which the first chunk reference points to are actually available in version k, thus may be obtained locally; (b) the second chunk reference, however, points to contents that the redundancy footprint of g does not cover, therefore B can no longer obtain them locally. To repair the second situation, B has no option but to

send a second request to *C*, asking for the plain transfer of the contents that the latter chunk reference points to.

In contrast to exploiting redundancy footprints at the sender side, the receiver side counterpart has a relevant network cost. In some cases, it imposes an additional round-trip to our, otherwise simple, deduplication protocol. This reveals a trade-off between more effective in-band redundancy detection and less round-trips to complete synchronization. Hence, depending on the actual system and workload, enabling exploitation of redundancy footprints at the receiver side may or may not compensate. In the remainder of thesis, we adopt the option of protocol simplicity and assume that dedupFS exploits sender-side footprints only.

6.4 Similarity Graph Maintenance

So far, we have introduced the chunk redundancy state of a site and described how we may exploit it for data deduplication. This section explains how a site dynamically maintains such a state as it discards old versions and stores new ones. The former occurs as the site prunes versions from a r-unit or decides (or is instructed) to cease replicating a given r-unit, discarding the corresponding replicas. The latter occurs as new versions are created locally or arrive from remote sites.

As a result, new similarity trees may appear; and existing trees may gain nodes, loose nodes (possibly changing their root chunk) or disappear. The system must, therefore, react to such changes and update its chunk redundancy state accordingly. Such a procedure may be slow; hence, it runs in background, and not synchronously with synchronization sessions.

6.4.1 Version Addition

We start by describing the technique we employ to detect chunk redundancy between new versions and previously stored versions, at a given site. It is important to note that our architecture supports a vast diversity of techniques for such a purpose. The sole requirements are that the chosen technique be able to output the chunk redundancy state used by our data deduplication protocol and not be prone to false positives. In particular, approaches such as the fixed-size sliding block method of rsync [TM98] (if adapted to not produce false positives) or diff-encoding [Mac00], among others [PP04, YK04], can easily replace the method that we propose as follows.

Our method divides new versions into variable-sized chunks using the content-based technique proposed by Muthitacharoen et al [MCM01]. A content-based approach minimizes the impact of *insert-and-shift* operations (i.e. operations that modify a file by inserting some bytes and shifting the remaining contents) on the chunk redundancy that is detectable between the newer and the original versions [MCM01].

For each version, we run a 48-byte sliding window along the contents of the

version. For every such an (overlapping) 48-byte region, we calculate a fingerprint. Similarly to [MCM01], we use Rabin fingerprints [Rab81] due to their efficiency when computing on a sliding window. When a pre-defined number of lowest-order bits of the fingerprint of the current region equal a pre-defined integer, we consider the region a chunk boundary. Accordingly, the current chunk ends at the boundary, and a new chunk starts at the next byte.

We may parameterize the expected chunk size by the number of lowest-order bits compared. A higher number of bits implies a lower probability of boundary selection, and thus a larger expected chunk size. More precisely, n lowest-order bits entail an expected chunk size of 2^n.[8] For the moment, we leave the value of n unspecified; Section 8.3 evaluates and discusses possible values of n.

The last region of a value implicitly constitutes a boundary. Furthermore, we impose a minimum and a maximum chunk length in order to prevent abnormal chunk sizes. More precisely, we ignore any chunk boundaries before the minimum size; while we always consider the first region that makes the maximum size of the current chunk as a boundary, independently of its fingerprint.

Having delimited a new chunk, we detect whether a chunk with similar contents is already locally stored or not. We avoid an exhaustive byte-by-byte comparison of the contents of every previously stored chunk with a help from compare-by-hash. We maintain a *chunk hash table* that associates hash values with references to actual chunks within the versions stored by the site. The chunk hash table only needs to comprise entries for either root chunks or literal chunks; it references no child chunks.

For each new chunk, we look up the chunk hash table for chunks with a matching hash. For each match, we compare the bytes of the new chunk with those of the equally-hashed chunk. In the case of a false positive match, we continue our search in the chunk hash table, as more than one equally-hashed, yet distinct chunks may exist. Therefore, we prevent data corruption due to potential hash collisions at the cost of byte-by-byte comparison when hashes match.

Figure 6.9 illustrates the process of chunk division and lookup for a new version, k, at some site A. The Rabin fingerprints calculated for each region of k yield three boundaries. Hence, we divide version's contents into four chunks, for which we obtain a hash value. Looking up each such hash value in the chunk hash table of A, we determine that the third chunk of the version has the same hash value as one chunk that A already stores (belonging to version $\langle A, 6\rangle$). We then compare the contents of both chunks, byte by byte, in order to check whether they are effectively redundant. The remaining chunks of k are necessarily literal at A, since no equally-hashed chunks exist in the chunk hash table.

Since hash collisions are no longer a critical event, we may increase the performance of hash computation by using a fast, non-cryptographic, hash function.[9]

[8] Assuming infinite version size. In practice, average chunk sizes will be lower due to chunks that end at end-of-file bytes, rather than at a boundary.

[9] Following the guideline suggested by Henson in [HH05].

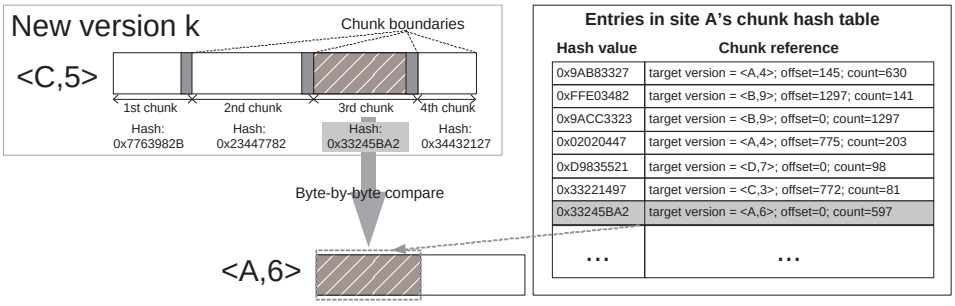


Figure 6.9: Example of chunk division and lookup for a new version, *k* that site *A* is about to store. Site *A* already stores other versions, whose root and literal chunks are represented in *A*'s chunk hash table.

More precisely, we recycle the Rabin fingerprints that chunk division already calculates for such a purpose. Our hash function consists of a 64-bit exclusive-or (XOR) of the Rabin fingerprints of each 48-byte portion of the chunk, which are already available for free (in terms of computational cost) from the chunk division phase.

The space requirements of the chunk hash table may take up a non-negligible portion of the space requirements of dedupFS if parameter *n* of the chunk division procedure is set for sufficiently small chunks. A solution for minimizing such an overhead is out of the scope of the paper. Possible solutions to alleviate it include the compressed representation of a chunk hash table proposed in the context of the figerdiff system [BJD06]; or the use of bloom filters [Blo70] to identify large groups of chunks, replacing their individual hashes. As an alternative option, the chunk hash table may be completely constructed and, afterwards, discarded each time the chunk redundancy step is executed. Notice that such a data structure is not necessary for the data deduplication protocol.

For each new version, we run the above-described chunk redundancy detection method. Such a method needs not run immediately once a new version exists, and may instead be postponed to idle-processor periods. However, synchronization sessions that may take place in the meantime will not be able to exploit any chunk redundancy that affects the new versions.

Whenever the chunk redundancy detection method gets a match for a chunk of a new version, we define it as a child chunk of the chunk referenced in the chunk hash table. More precisely, i) we add a reference to the chunk referenced in the chunk hash table to the chunk reference list of the new version, and ii) we add an entry to the similarity table, pairing the new version to the older value. If, otherwise, no identical chunk is found, then we simply add a chunk reference to the new chunk in the chunk hash table.

We illustrate the above steps by recalling the previous example, in Figure 6.9. Let us assume that the third chunk of version *k* (identified by $\langle C,5\rangle$) has the same contents as the chunk at version, and thus byte-by-byte comparison detected such

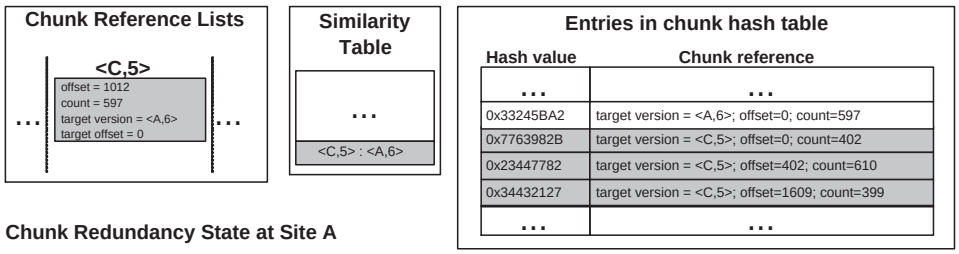


Figure 6.10: Chunk redundancy state of A after A receives and stores version k from the example in Figure 6.9. Changes to such each data structure are in highlighted in gray. Version k is identified by $\langle C,5\rangle$. One of its chunk is literal with a chunk in version $\langle A,6\rangle$, which A already stores; all k's remaining chunks are literal at A.

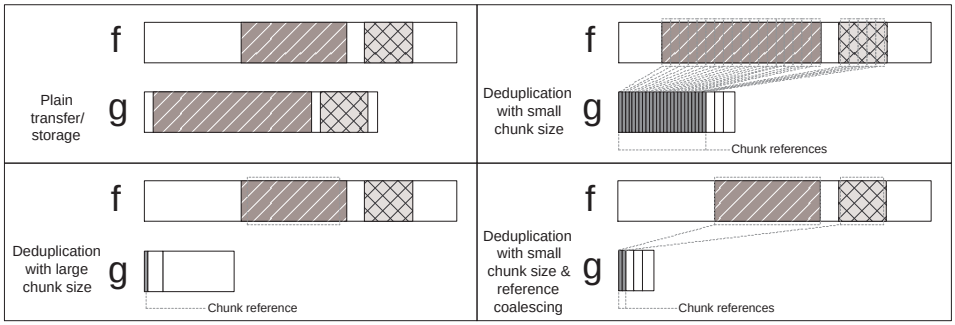


Figure 6.11: Example of effective volumes in storage or synchronization of version g with different approaches: plain storage/transfer, and deduplication with a large expected chunk size, small expected chunk size, small expected chunk size with reference coalescing. Large chunk sizes yield lower precision in redundancy detection, while smaller chunk sizes yield higher detection precision but may have a higher space overhead due to chunk references to chunks that are contiguous at the target version (v). Reference coalescing achieves the accuracy of small chunks, at acceptable chunk reference overheads.

a redundancy relation. Recall that the remaining three chunks of k are literal at A. Figure 6.10 depicts the changes to the chunk redundancy state of A after it receives and stores version k. The chunk reference list of k includes a single entry, which points to the chunk at version $\langle A,6\rangle$, with which k's only redundant chunk (the third one) shares its contents. Hence, the latter chunk becomes becomes a child chunk of the former. Accordingly, the similarity table of A now holds a new entry denoting that k has a child chunk of a root chunk in version $\langle A,6\rangle$. Finally, the chunk hash table now includes entries that reference each literal chunk of k.

6.4.2 Reference Coalescing and Chunk Size Trade-Offs

We complement the above procedure with a simple step of *reference coalescing*. Before adding a new chunk reference to the chunk reference list of the new version, we check if the chunk reference points to a post-contiguous portion of the

previous reference in the list. If so, then we simply modify the previous reference, by incrementing its length field with the length field of the new reference.

We illustrate the advantage of reference coalescing with an example, in Figure 6.11. Reference coalescing retains the memory overhead of the chunk reference lists at a low level, even if average chunk size decreases. Chunk redundancy detection can be made more aggressive by decreasing the expected chunk size, which increases the probability of redundancy detection. Apart from pathological exceptions (which we address in Section 8.3.2.1), reference coalescing ensures that decreasing chunk sizes (down to nearly the size of a chunk reference) will monotonically decrease the aggregate size of literal chunk contents plus chunk reference lists. In fact, if a large chunk was detected before, then subdividing it into a consecutive set of smaller chunks will also result in a single chunk reference, as originally. In fact, new chunk references will (coalesced) typically only arise when small redundant chunks, which were previously hidden inside larger literal chunks, become detectable.

Hence, the aggregate size of literal chunk contents will decrease more rapidly than chunk reference lists will grow, due to reference coalescing. From the viewpoint of the data deduplication protocol, such an aggregate size is very important, as its components directly affect the bandwidth efficiency of the protocol. There is, however, a trade-off if we regard the negative consequences of smaller chunk sizes on the memory requirements of the chunk hash table and to the performance of the chunk redundancy detection procedure. Nevertheless, it is worth noting that, if we exclusively focus on the time and bandwidth efficiency of the data deduplication protocol, such a consequence is irrelevant. Namely, the data deduplication protocol does not use the former structure, and the latter procedure is run off-line, rather than during synchronization.

6.4.3 Version Removal

Version removal (due to log pruning or replica discard) implies removing the chunks of a given set of versions from the chunk redundancy state of the site. For some version, v, to remove, it takes the following two steps to accordingly update the chunk redundancy site.

A first step moves any root chunk that v may hold to some other version that holds (at least) one child of the root chunk. We obtain, from the similarity table, the list of versions that have at least one chunk reference pointing to v. For each version v' in such a list, we check which of its chunk references point to a chunk in v. If we find one such reference, we choose it as the new root chunk. Accordingly, we (i) erase the latter chunk reference from the chunk reference list of v'; (ii) change the chunk reference in the hash table to point to the new root chunk; and, (iii) if the version that includes the new root chunk, v', does not store the chunk's contents (due to redundancy-compression, as Section 6.3 describes), we insert the chunk contents (from v) in v', at the corresponding position among the chunks with contents already stored in v'. We continue the search for references to chunks in

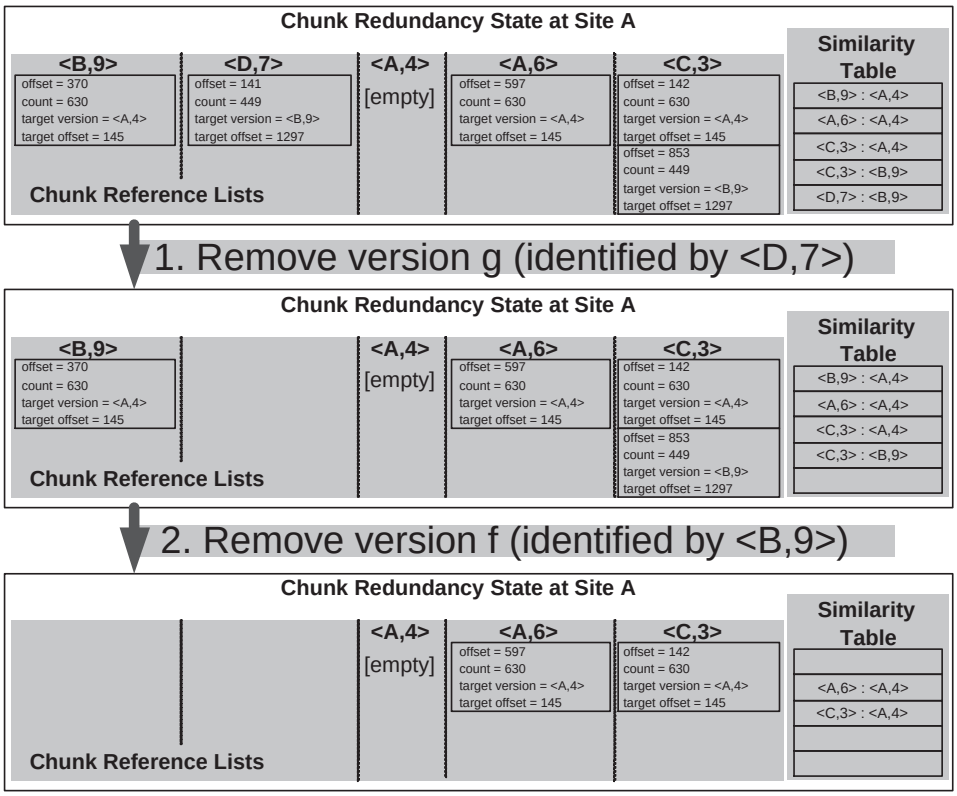


Figure 6.12: Example of changes to chunk redundancy state after successively removing versions g and f from A. Version g has one redundant, child chunk (besides other literal chunks), while f has both child and root chunks.

v. If we find references to chunks in v for which we have already performed the above steps, and chosen a new root chunk, then we simply change it to point to the new root chunk.

Finally, a second step removes any entry in the similarity table that refers to v.

We illustrate the above steps with the example that Figure 6.12 depicts. In the example, site A starts by removing version g, identified by $\langle D, 7\rangle$. By inspecting the initial chunk redundancy state of A, g has no root chunks, as no other version references any chunk of g. Hence, removing g is straightforward: we erase its chunk reference list, the entries from the similarity table that have k as the source version and the entries in the chunk hash table that target g (not depicted in Figure 6.12). When A removes version f (identified by $\langle B, 9\rangle$), however, we further need to move the single root chunk that f holds to some other version that holds one of its child chunks. In this case, the only version that has a child chunk of f's root chunk is $\langle C, 3\rangle$ (as we can see from the similarity table). Hence, we erase the corresponding chunk reference from the chunk reference list of $\langle C, 3\rangle$; update the chunk hash table to point to the new root chunk (not depicted in Figure 6.12); and,

if $\langle C, 3\rangle$ is redundancy-compressed, copy the contents of its new root chunk from the stored contents of f to $\langle C, 3\rangle$.

Contrary to the case of new versions, a site can only prune version v after after the above steps complete. Otherwise, the content of any root chunks in v would be discarded without checking if any of its child chunks had the chunk's contents available. If every child chunk was redundancy-compressed, then immediately discarding the root chunk would result in content loss.

6.5 On Out-Of-Band Redundancy

Compare-by-hash has the distinctive property of being able to detect redundancy among *any* pair of chunks. In turn, dedupFS is only able to detect in-band redundancy. Chunk redundancy between a pair of chunks where none of them belongs to the common version set of the communicating hosts escapes from dedupFS's redundancy detection scheme. This seems to be an unavoidable limitation of any solution that follows our approach.

As we discuss previously and confirm with experimental results in Section 8.3.4.1, out-of-band is close to negligible in a very representative set of workloads. Under such a condition, the advantages of dedupFS over compare-by-hash are effective.

Nevertheless, in a system with strong sources of out-of-band redundancy, dedupFS may no longer be more efficient than compare-by-hash. We distinguish three main sources of such out-of-band redundancy, which we exemplify as follows:

1. Non-copy transformations on common data. As an example, consider two concurrent versions, v_1 and v_2, of a given source code project, sharing numerous redundant chunks with a common earlier version, v_0. Host A stores v_1 and v_0, while host B stores v_2 and v_0. Clearly, the transference of either v_1 or v_2 to B or A, respectively, would perform efficiently in dedupFS, as it would be able to detect the existing chunk redundancy. Now, suppose both hosts compile their versions, obtaining the corresponding sets of binaries, b_1 and b_2, respectively. If the compiler and the execution environment are the same, b_1 and b_2 will also share significant similarity, a direct consequence of the similarity between their source code versions. However, dedupFS can not detect such a similarity.

2. Data propagation through external sources. If two beginner programmers, working on separate projects, access a common web page containing code examples, then it is possible that both will copy&paste similar code portions from the web page. In theory, both projects share similarities with a common version (the web page). However, dedupFS is not able to infer that.

3. Common application-produced contents. Two binary executable files, produced by the same compiler, may share similar headers. These do not consist of similarities due derived from data versions, but instead consist of a common pattern due to the usage of a common application.

A careful analysis of the sources of out-of-band redundancy in a particular system is needed to determine whether dedupFS is, in fact, a better option than compare-by-hash in such a system. Evidently, if the concerned users and applications impose a solution that resists hash collisions, then the second solution is not a valid option.

6.6 Summary

The storage and synchronization of replicas whose updates use a state-based representation may introduce important memory and network overheads. Such overheads may easily render the replicated system inadequate for operation in environments where memory and bandwidth are limited resources.

We propose a novel technique for data deduplication in distributed storage systems. Our solution is able to eliminate two crucial limitations of data deduplication through compare-by-hash. Namely, we (i) eliminate the possibility of data corruption due to hash collisions; and (ii) we impose a significantly lower protocol overhead, both in terms of meta-data sent across the network and message round-trips; (iii) while detecting redundant volumes that are, at least, comparable to compare-by-hash, for most workloads we expect. We have implemented a distributed archival file system, dedupFS, which employs our solution.

Our approach follows the principle that, if a sender host is able to determine which of its local versions are also stored at the receiver host, then the sender may detect chunk redundancy by simply comparing the contents to send with those of the common set of versions. Such a principle has always lied underneath older approaches for data deduplication, such as Delta-Encoding and Cooperating Caches, though not exploited to its full potential. In this sense, our contribution can be seen as an enhancement over such older approaches. It enlarges the set of detectable chunk redundancy, hence attaining higher data deduplication efficiency, by a novel combination of modern techniques. Namely, vector sets, content-based indexing and reference coalescing.

Chapter 7

Implementation

We have implemented two prototypes incorporating the contributions that the previous chapters propose. A first project consists of a simulator of a distributed replicated system. The simulator runs a set of mobile and stationary sites, each holding replicas and applications that modify the latter. It allows a thorough evaluation of generic commitment protocols under a wide range of network and workload conditions. We have used the simulator to evaluate implementations of the commitment protocols in Chapter 4, as well as the extension of such protocols that Chapter 5 proposes.

For the contributions on efficient state-based replica storage and synchronization (Chapter 6), we have implemented a prototype of a concrete replicated system. More precisely, it consists of a replicated file system, called dedupFS, a representative example of a state-based system. Here, the focus is on evaluating replica storage and synchronization, and not on the commitment protocol. Therefore, the prototype runs a replication protocol that has no commitment protocol.

The following sections detail such implementations, starting by the simulator in Section 7.1, and then proceeding to the replicated file system in Section 7.2. Finally, Section 7.3 summarizes the chapter.

7.1 Simulator

The simulator was implemented in C# and consists of a Windows application. Figure 7.1 presents the user interface. The simulator includes 17 classes, and more than 5000 lines of code.[1]

[1]The source code of the simulator project, including the protocols and simulator described in the following sections is publicly available at `http://www.gsd.inesc-id.pt/~jpbarreto/VVWVSim.zip`.

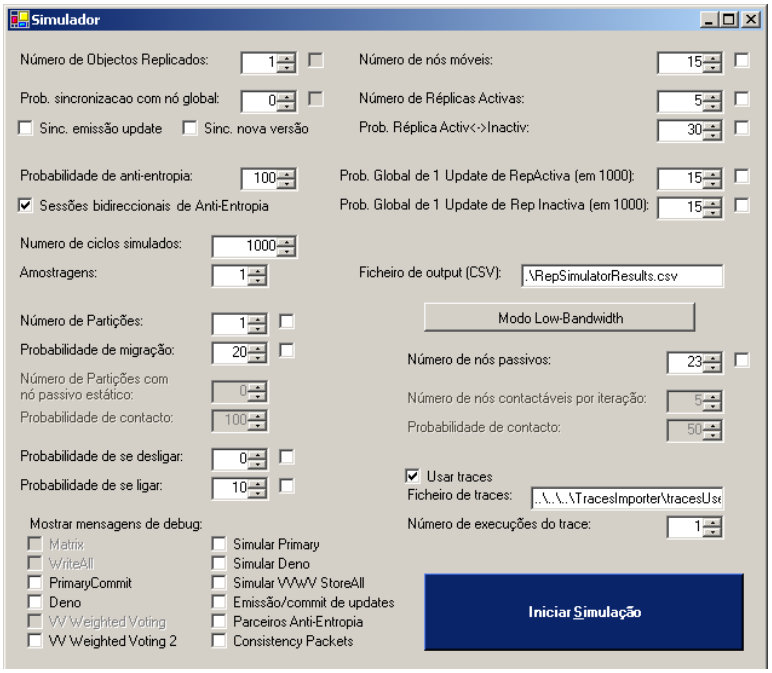


Figure 7.1: Snapshot of the simulator's graphical user interface.

7.1.1 Base Classes

The simulator includes a collection of mobile sites, implemented by class `Site`. Sites are randomly distributed by a set of network partitions. The simulator assumes a set of objects, each of which may be replicated at a subset of sites. For each object, a site may either hold a replica or not. A non-replica site may be willing to carry consistency packets, in which case we designate it a non-replica carrier (*NRC*).

Class `Replica` implements replicas. Each replica runs one or more commitment protocols. Any commitment protocol is a subclass of the abstract class `ConsProtocol`. Among others, `ConsProtocol` provides two important public (abstract) methods to the replica: `issueUpdate` and `pullReplica`. The replica calls the former method to notify that a new update has been issued. The latter method is called when synchronization occurs, requesting the local protocol instance to pull update data from a remote commitment protocol instance, running on another replica. Furthermore, in order to support the exchange of consistency packets, `ConsProtocol` has the public methods `genConsistencyPacket` and `applyConsistencyPacket`; which respectively generates a consistency packet (class `ConsPacket`) reflecting the current replica state, and receives a consistency packet, incorporating its meta-data into the replica's state.

numSteps	Number of time steps of each simulation.
numSamples	Number of simulations to run.
numObjs	Number of objects.
numSites	Number of mobile sites.
numPart	Number of network partitions.
mobProb	Probability of a site migrating to a different partition.
numReplicas	Number of replicas per object.
numActive	Number of active replicas per object.
probUpdHigh	Update probability of active replica.
probUpdLow	Update probability of inactive replica.
activateProb	Activation probability.
failProb	Failure probability of a site.
resumeProb	Probability that a failed site resumes operation.
bidirAE	Bidirectional anti-entropy if true, unidirectional if false.

Table 7.1: Simulation parameters.

7.1.2 Update Commitment Protocols

We have implemented a number of relevant commitment protocols, as subclasses of `ConsProtocol`. A first one is our proposed protocol, `VVWV`. For a comparative evaluation of VVWV, we have also implemented relevant state-of-the-art commitment protocols. Namely, the primary commit protocol (`Primary`) and Deno's basic weighted voting protocol (`BasicWV`). In the case of Deno, the implementation extends the original protocol specification with optimistic replication support.

Every commitment protocol class implements the methods `issueUpdate` and `pullReplica`, for regular operation. Only `VVWV` supports the exchange of consistency packets, implementing the `genConsistencyPacket` and `applyConsistencyPacket` methods. It exchanges consistency packets with the protocol's specific meta-data, instances of class `VVWVConsPacket`, a subclass of `ConsPacket`.

7.1.3 Simulation Cycle and Parameters

Table 7.1 enumerates the variables that parameterize each simulation. The simulator divides time into logical time steps. At each time step, each site *A*:

1. With a given mobility probability, *mobProb*, *A* migrates to a different, randomly chosen, network partition;

2. If *A* is replica or NRC, then *A* randomly selects a site, *B*, from the set of sites in *A*'s current partition. For each object, *A* either performs anti-entropy from *B* (if *A* and *B* are replicas); or receives a consistency packet from *B* (if *B* and/or *A* are NRCs);

3. If *A* is a replica, *A* generates an update with a given update probability.

Anti-entropy may be either unidirectional or bidirectional; parameter *bidirAE* determines such an option. Each replica may be active or inactive. A replica's update probability depends on whether the replica is active (*probUpdHigh*) or inactive (*probUpdLow*). An inactive replica exchanges, with a given activation probability, *activateProb*, its inactive status with an active replica after pulling anti-entropy information from it. The differentiation between active and inactive replicas allows us to simulate non-uniform update models; in particular, the hot-spot model [RRP99], which assumes, based on empirical evidence, that updates occur with greater probability in a small set of replicas.

Furthermore, each site may fail temporarily at the beginning of a time step, with a given failure probability, *failProb*. A failed site neither issues any updates nor synchronizes with other sites. A failed site may resume its operation at a subsequent time step, recovering the previous state, with a given resume probability, *resumeProb*.

Each simulation runs the above phases for a number of time steps, outputting results from such an execution. In the case of multiple protocols, they run side-by-side during each execution of the simulator; this enables a fair comparison of each protocol. For more statistically meaningful results, the simulator repeats each experiment a number of times (defined by *numSamples*); each such experiment is different, as it uses distinct randomization seeds. The simulator then outputs average results, as well as individual results. For evaluation of ranges of parameter values, a bath simulation option is available. A batch simulation automatically runs multiple simulations, traversing the range of values that are specified for a subset of the parameters.

Finally, one may replace the random mobility and partitioning of sites by submitting a mobility trace to the simulator. A mobility trace consists of a list of tuples, each specifying an encounter, at a given moment, between two sites. The mobility trace is an ASCII text file, where each line represents one tuple. For instance, the following line denotes an encounter between sites 12 and 18 at a given moment in time:

```
2003-11-20 18:08:52 12 18
```

Mobility traces allow the emulation of mobility and partitioning traces observed from real systems. When a mobility trace is specified, all sites exist in a single partition, and anti-entropy occurs only due to encounters specified in the trace file.

7.1.4 Measures

Each commitment protocol notifies the simulator of commitment-related events. These include an update becoming stable, aborting, or committing. Furthermore, the simulator may inquire for memory and bandwidth consumption with protocol-specific control data (e.g., version vectors).

From such information, the simulator is able to measure, for each replica, the average, minimum and maximum values of:

1. Number of stable, aborted and committed updates;

2. Number of updates that became stable, aborted or committed after regular anti-entropy; and number of updates that became stable, aborted or committed after reception of a consistency packet;

3. Stability and commitment delays, which consist of the number of time steps between the step where an update was issued and the step where it became stable and committed, respectively;

4. Memory and network consumption with control data;

5. Dimension and number of consistency packets in the local buffer.

The above results are output as a file in the Comma-Separated Value Format (CSV) [Sha05].

7.2 dedupFS

We have implemented dedupFS as a file system for the Linux operating system.[2] dedupFS complements the basic file system services with a versioning service. Complementarily to the current version of each object (file or directory), it maintains and offers access to older versions of such objects. Versioning is an important feature for many users and applications for a number of reasons [SGSG02]. They allow recovery from user mistakes or system corruption [SFH+99], backup [CN02, QD02, SGMV08], post-intrusion diagnosis [SGS+00], auditing for compliance with electronic records legislation [PBAB07], or versioning support for collaborative work [C+93].

More than a local versioning file system, dedupFS supports distributed collaboration through sharing of files and directories. Groups of users may share subsets of their file-spaces. dedupFS enables applications to create new shared objects, as well as read from and write to them in an optimistic fashion. dedupFS runs a replication protocol that ensures synchronization of shared object replicas that each site holds.

dedupFS is an instantiation of the generic replication service that the present thesis addresses, where replicated objects consist of files and directories. Being a file system, it is a concrete example of a state-based replicated system; here, state-based updates consist of file versions. Hence, the implementation of dedupFS allows a realistic evaluation of the contributions for efficient state-based replica storage and synchronization, which Chapter 6 proposes. As the next sections explain, dedupFS incorporates such contributions in order to attain space-efficient file system versioning, and network-efficient synchronization.

[2] The source code of dedupFS is publicly available at:
`https://www.gsd.inesc-id.pt/~jpbarreto/dedup.zip`.

The focus of implementing dedupFS is on evaluating replica storage and synchronization, and not on the commitment protocol. Therefore, the following sections describe a system with no commitment protocol.

The remainder of the section is organized as follows. Section 7.2.1 starts by presenting the implementation details of the basic file system, including those concerning version tracking and distributed collaboration features. Section 7.2.2 describes the local similarity detection process. Section 7.2.3 then describes the mechanisms for efficient replica synchronization.

7.2.1 Base File System

dedupFS relies on the Filesystem in Userspace (FUSE) kernel module, version 2.6.3. FUSE provides a bridge between dedupFS's code, which runs in user space, and the actual Linux Virtual File System kernel interfaces. The choice of FUSE allowed for rapid prototyping of dedupFS; namely by allowing user-level debugging and significantly quicker unmount-compile-mount cycles. As a trade-off, the performance of the FUSE-based prototype is worse than would be achievable with a traditional kernel-level file system. However, as Chapter 8.3 later discusses, such a performance degradation does not affect the aspects that we are interested in evaluating in the protocol; that is, those regarding our contributions.

Any non-privileged users may mount dedupFS at a given mount point, as well as unmount it. dedupFS supports a large set of interfaces from the POSIX file system API, which is sufficient for most legacy applications to correctly use dedupFS to store and access files and directories.

In its current form, dedupFS does not store data directly on disk. Instead, it uses the native file system to as an lower-level storage layer. The native file system stores dedupFS's state in the form of files and directories. Such a choice imposes additional performance overhead due to the extra layer, and has the problem of leaving dedupFS's state exposed to users that have access to the native file system. Again, the need of rapid prototyping justifies such a choice. A future version should implement direct disk access.

dedupFS's unit of storage is the i-node, identified by a unique version identifier, as defined in Section 6.1.1. Every directory and file version has one corresponding i-node that stores its data. Each i-node, in turn, is stored as a file in the native file system, whose name is a string with the corresponding version identifier. For instance, a file version created by writer 2 and assigned a counter value 3 is stored at file `"3_2"` of the native file system directory.

For directories, we adopt a similar solution as in the Elephant file system [SFH+99]. A directory is stored in a single i-node, identified by the directory version, which comprises a directory entry for each object in the directory. Changes to the directory (e.g., objects added, removed, or attributes changed) result in appending a new entry to the directory i-node and setting the `active` flag of the previous entry corresponding to the affected object to false. As files and directories, directory entries are also assigned a version identifier, stored along the entry. At a given

moment, the contents of a directory comprise the active entries in its i-node.

Files are identified by the identifier of their first version. Directory entries store such an identifier, which remains constant throughout the file's lifetime. When opening a file, located at a given path, we need to be able to map the file identifier (obtained from the directory entry corresponding to the path) to the identifier of its current version. dedupFS maintains a version information table, indexed by file identifier, which contains, for each file: version identifier, size, compression mode (plain, redundancy or reference), and `hasRefs` flag (explained below) for the current version of the file; and the i-node identifier of the file log (if any). dedupFS stores the version information table as a file in the native file system. By looking up such a table, dedupFS is able to open and correctly retrieve the contents of the current version of a file.

If a file has only a single version, it has no log i-node. This is an important storage optimization since many files of a file system are never changed. If, otherwise, new versions are added to a log-less file, then dedupFS creates a new i-node to the log (and assigns a version identifier to it, as well), updating the file's entry the version information table. The log consists of a list of version information entries, similar to the ones in the version information table. Such entries allow access to the archived versions. Currently, dedupFS implements on-close version creation. It may be easily extended to support other policies [SFH+99].

For general i-node caching, dedupFS transparently relies on the cache of the native file system. However, for improved directory lookup, a frequent performance-critical task, dedupFS needs to prioritize caching of directory entries. Since dedupFS cannot prevent the native file system cache from blindly replacing older directory i-nodes with lower-priority i-nodes, dedupFS maintains an independent in-memory directory entry cache. The cache adopts a FIFO replacement policy.

7.2.2 Local Chunk Redundancy Detection

As new versions are added to the system, they are added to a list of unchecked versions. A user command explicitly initiates chunk redundancy detection across the unchecked versions and the remaining locally stored versions.[3]

We maintain an in-memory chunk hash table. We garbage collect entries of the chunk hash table by maintaining a reference counter along each entry. When chunk redundancy detection starts, it divides unchecked versions in chunks, checks whether similar chunks already exist in the chunk hash table. If so, dedupFS uses the information in the chunk hash table entries and adds a new chunk reference to the list of the version, incrementing the reference counter of former entries. If not, dedupFS adds entries to the chunk hash table, which point to the respective non-redundant chunks.

[3]The choice of an explicit command for redundancy detection was motivated for increased control during the experimental evaluation of dedupFS. Naturally, a release version of dedupFS would have automatic redundancy detection in background.

Chunk reference lists are stored at the head of a version, prefixed by a short integer that indicates the length of the list. If a list has no elements, then such a head is omitted. Each element takes up 20 bytes. dedupFS determines whether it should look for a chunk reference list or not by consulting the `hasRefs` flag in the version information entry of the version.

Chunk division and hashing code is based on the source code of LBFS [MCM01], modified in order to use the XOR of Rabin fingerprints as chunk hashes, instead of the original SHA-1 hash function. Chunk matches are complemented with byte-by-byte comparisons between the contents obtained from the i-nodes of the matching chunks, in order to avoid false positives due to hash collisions. For evaluation purposes, we have also implemented a SHA-1 mode, which computes and compares SHA-1 chunk hashes instead.

We implement the similarity table as a file that stores a variable sized array of version identifier pairs. Therefore, look up of such a structure is linear with the number of stored pairs. A perhaps better option consists of implementing the similarity table as a disk-resident hash table, as proposed by Quinlan and Dorward [QD02]. We plan to study such an option for a future version of dedupFS.

7.2.3 Synchronization

Site communication is supported by SUN RPC [Sri95] remote procedures over UDP/IP, according to the protocol that Section 6.2.2 describes. Currently, dedupFS does not support multiple r-units; instead, every object is included in a single, system-wide r-unit.[4]

After receiving a request for synchronization, the sender site replies with the new directory entries and new file version meta-data, including the references for the redundant chunks in such versions. The remaining non-redundant chunk contents are sent over a TCP/IP socket, which is established between both sites immediately after the RPC request is made.

For evaluation purposes, we have also implemented an alternative synchronization protocol, which uses the compare-by-hash protocol of LBFS for content transference. This mode is used with the SHA-1 mode of local chunk redundancy detection.

7.3 Summary and Discussion

We have implemented two prototypes incorporating the contributions that the previous chapters propose. A first prototype is a simulator of a distributed replicated system. It allows an exhaustive evaluation of generic commitment protocols under a wide range of network and workload conditions. It allows us to evaluate implementations of the commitment protocols in Chapter 4, as well as the extension of such protocols that Chapter 5 proposes.

[4] A multiple r-unit version is, naturally, planned for future work.

A second prototype is a functional archival, distributed file system for Linux, called dedupFS. dedupFS is a state-based optimistically replicated system that incorporates the versioning-based data deduplication scheme that we propose in Chapter 6. The implemented system enables evaluating of our data deduplication scheme with real file usage workloads, as well as comparing it with other existing solutions for file synchronization and storage.

The next chapter presents and discusses the experimental results that the above implementations produced.

Chapter 8

Evaluation

This chapter evaluates the contributions that the previous chapters introduced. Section 8.1 starts by evaluating VVWV. We use the replication simulator to compare VVWV against other epidemic alternatives, namely epidemic primary commit and Deno's epidemic weighted voting protocol. Section 8.2 then evaluates the benefits of complementing VVWV with decoupled agreement by exchange of consistency packets with non-replicas. Again, we resort to results from the simulator for such an analysis.

Section 8.3 then evaluates our data deduplication solution for replica storage and update propagation. Such an evaluation is based on real executions of the dedupFS implementation, using practical workloads.

Finally, we summarize our conclusions in Section 8.4.

8.1 Version Vector Weighted Voting Protocol

This section evaluates the commitment efficiency of VVWV, which Section 4.2 described. For the moment, we focus only on the protocol, and do not consider its agreement decoupling using consistency packet exchange. For most of the aspects that we evaluate, we compare VVWV with two representative solutions for epidemic update commitment. A first one, which we designate as *Primary*, uses a primary commit approach (similarly to Bayou [PST+97]). A second one consists of Deno's epidemic weighted voting approach, which requires one election to commit each individual update.

Similarly to VVWV, all solutions are semantic-oblivious. We chose not to include semantic solutions due to the lack of reference results, benchmarks or simulation models concerning the commitment efficiency evaluation of such approaches in related literature (e.g. [DPS+94, TTP+95, PSTT96, KRSD01]). Furthermore, the present evaluation omits non-epidemic commitment solutions such as traditional quorum systems for commitment in strongly-connected distributed database systems [AW96]. Our focus is on weakly connected environments, where such traditional approaches are not appropriate, as discussed elsewhere [HSAA03, Kel99].

The present evaluation does not consider permanent site failures, only temporary partitioning, and hence does not study protocol availability. We justify such a decision by the limited meaning that measurements from finite experiments have to characterize availability; and by the fact that we have already compared the availability of all three protocols on a theoretical ground, in Section 4.1.3. Therefore, the conclusions that we draw in this chapter do not take into account the crucial limitation in availability of Primary (which depends on a single point of failure) when compared to the weighted voting solutions.

We have obtained all results concerning VVWV in Store-One mode (see Section 4.2.3), as we have observed that the difference in measurements from the Store-All mode is negligible. We fundament such an observation with experimental results in Section 8.1.2.

In the following analysis, measurements are obtained from executions of the replication simulator that Section 7.1 described. For coherence and fairness of our comparative results, the underlying simulation model is based on the same model adopted by the reference work on epidemic commitment protocols by Keleher et al, in the context of the Deno replicated system [Kel99]. We extend their basic simulation model with additional parameters and measures, which Section 7.1 described. Such an extension enables a richer analysis under a larger universe of networking and workload scenarios than those that Deno's evaluation considers.

Consistently with the central concerns of the present thesis, we are mainly interested in using experimental results to answer the following questions:

1. How much time does it take for an average update to be committed?
2. What is the proportion of updates that are effectively committed, and not aborted?

Furthermore, not only are we interested in studying the above aspects in different networking conditions (e.g. varying the level of partitioning of our system, and the mobility of sites in transit between partitions), but also in different workloads (e.g. varying the rate at which applications issue updates, and how active replicas are distributed among the system).

By default, we consider a system of five replicas, which we consider representative of the scenarios that the thesis addresses. Furthermore, all replicas are active by default; for those experiments where we consider a lower number of active replicas, the activation probability is 30%. A global update probability of 5%, evenly divided by the active replicas (i.e. $updProb = \frac{0.5\%}{numActive}$), was considered.

Unless noted in the text, the variation of the previous parameters was found to not have a relevant impact on the obtained results and, most importantly, not to affect the conclusions that we draw from the results. For those measures on which the variation of some of the previous parameters does have a relevant impact, we present results for a varying range of values of such parameters.

We adopt the following procedure for obtaining the measurements on which our evaluation is based. We consider samples comprising measurements from five

simulation runs. Each run lasts for 2000 logical time steps and, before each run, we initialize the simulator with a distinct randomization seed, so that each of the five runs is effectively distinct. The results we present next are an average of the measurements taken from the five runs.

8.1.1 Average Agreement Delays (AAD)

Many applications using an optimistic replication system will tolerate accessing a weakly consistent tentative value for some of their operations, but will wait until their tentative work is finally incorporated into the strongly consistent value before proceeding with some critical operations. The delay separating both moments is, naturally, a crucial aspect to the effectiveness of a commitment protocol.

One way to evaluate such an aspect is to measure the agreement delay of a commitment protocol, which reflects the protocol's commitment and abort delays. We define commitment delay for an update, *u*, at some replica, *r*, as the number of time steps separating the step where some replica issued *u* and the step where *r* committed *u*. When *u* aborts, rather than committing, we reason about its abort delay. We define the abort delay of *u* at *r* as the number of logical time steps separating the issue of *u* and the first step where *r* commits a concurrent update (by happens-before) with *u*. Note that this definition does not require *r* to actually receive *u* and then abort it; it is sufficiently general to also consider the cases where *u* does not even propagate to *r*, because *r* has already (implicitly) aborted *u* by committing a concurrent update. Finally, the agreement delay of an update is either its commitment and abort delay, depending on whether the update commits or aborts.

A first experiment studies the average agreement delays (AAD) of *Primary*, *Deno* and *VVWV* under different networking conditions, both in terms of number of partitions and of mobility probabilities. Figures 8.1 to 8.3 present the results, distinguishing two mobility scenarios (low mobility, with *mobProb*=25% and high mobility, with *mobProb*=75%) over an increasing number of partitions. Each figure considers a decreasing number of active replicas; this way, we are able to analyze different update models. In the experiments with three and one active replica, we set the activation probability to 30%. Naturally, AAD is not constant across replicas. Hence, each figure depicts the AADs at each replica, in increasing order of individual AAD.

8.1.1.1 Analisys of AAD

As expectable, AADs increase significantly as the number of partitions hinders replica accessibility. Such an increase may be of almost one order of magnitude as we depart from a single-partition situation to 11 partitions with low mobility. High mobility of replicas strongly reduces the effect of higher partitioning, typically accelerating agreement to less than half the delay with low mobility.

We start by comparing the AADs of Primary and VVWV (we proceed shortly to compare VVWV with Deno). Table 8.1 presents the relative increase in AAD as

# Partit.	ith Replica to Agree 1st	3rd	5th	ith Replica to Agree 1st	3rd	5th	ith Replica to Agree 1st	3rd	5th
1	-4,5%	-8,9%	3,0%	-3,8%	-10,4%	-2,3%	-3,6%	-14,9%	-0,9%
11	23,7%	-5,8%	-7,7%	16,9%	-4,7%	-7,9%	14,6%	-12,9%	-14,2%
21	76,7%	17,5%	4,1%	93,7%	15,9%	6,8%	5,8%	-10,0%	-13,7%
31	73,3%	14,3%	1,9%	59,5%	3,6%	-1,0%	-7,7%	-21,7%	-20,5%
41	152,6%	21,8%	1,8%	65,5%	13,7%	-1,2%	21,6%	-17,4%	-19,0%
	All Replicas Active			Three Active Replicas			One Active Replica		

Table 8.1: Relative increase in AAD from Primary to VVWV (i.e., $\frac{AAD_{VVWV} - AAD_{Primary}}{AAD_{Primary}}$), with low mobility, for different network settings.

one changes from Primary to VVWV, which we obtain from the results in Figures 8.1 to 8.3. (Positive percentages denote an increase when one compares the AAD of VVWV with the AAD of Primary.) With all and three replicas active, *Primary* generally performs substantially better than *VVWV* for the first few replicas[1]. Furthermore, the advantage concerning the first replicas to agree tends to grow as one weakens connectivity (by increasing the number of partitions and lowering replica mobility[2]) and as one increases the number of active replicas.

For a higher replica order, however, the AADs of *Primary* converge with and, in some cases, become higher than the AADs of the weighted voting alternatives (VVWV and Deno). Such a behavior becomes more evident as one considers larger systems. To illustrate, Figure 8.4 shows the AADs for systems with 6 to 10 replicas in the particular setting of 31 partitions, low site-mobility and all replicas active. In such larger systems, there always exists a replica order after which *Primary* is outperformed (in AAD) by the weighted voting protocols.

This observation has already been documented in the context of the Deno replicated system [CKBF03]. Keleher et al. explain it by the fact that, although the *Primary* is able to commit rapidly (since it suffices for the update to arrive at one replica, the primary), such a decision propagates relatively slowly to other replicas. This is because all other replicas must learn of the commitment, directly or indirectly, from the primary replica. In contrast, *Deno* and *VVWV* enable distinct replicas to either learn the decision from other replicas (as in the case of *Primary*), or decide the update independently by receiving sufficient votes.

Interestingly, in the case of the voting solutions (*Deno* and *VVWV*), the first small group of replicas (2 replicas in the 5-replica system) exhibit relatively similar AADs, while the remaining delays seem to approximate a linear function on the order of replicas. Such an effect is much less pronounced with *Primary*.

Of course, an important aspect is to compare both epidemic weighted voting protocols. Table 8.2 shows the relative increase of the AADs of Deno, when com-

[1]The exception being the single partition case. However, taking into account the relatively low AADs of this case (under 2.2 time steps, on average), when compared to the measurement precision (of one logical time step), we do not consider this exception as statistically meaningful.

[2]This observation is not deducible from Table 8.1.

#	ith Replica to Agree			ith Replica to Agree			ith Replica to Agree		
Partit.	1st	3rd	5th	1st	3rd	5th	1st	3rd	5th
1	0,9%	0,6%	0,5%	0,2%	0,4%	0,0%	0,5%	0,7%	0,1%
11	4,0%	2,4%	1,3%	4,3%	3,0%	2,2%	26,9%	24,5%	16,2%
21	8,6%	5,6%	3,3%	16,7%	12,6%	5,1%	75,0%	59,3%	35,3%
31	12,8%	8,0%	4,5%	29,8%	25,2%	13,5%	79,4%	64,2%	41,9%
41	16,3%	10,9%	3,8%	8,1%	5,3%	2,2%	76,4%	56,6%	27,0%
	All Replicas Active			Three Active Replicas			One Active Replica		

Table 8.2: Relative increase in AAD from VVWV to Deno (i.e., $\frac{AAD_{Deno}-AAD_{VVWV}}{AAD_{VVWV}}$), with low mobility, for different network settings.

pared with VVWV; again, we derive the values from the results in Figures 8.1 to 8.3.

As expectable, the AADs of *VVWV* become increasingly lower than those of *Deno* as connectivity degrades with partitioning and/or low mobility. This is the natural consequence of the higher frequency of situations of multiple-update candidates. A narrower set of active replicas also contributes to such an advantage of VVWV over Deno, for the same reason. Let us consider the case where all replicas are active. Starting with a single partition, where both protocols achieve practically similar AADs, the 3rd replica of Deno to agree takes 2.4%, 5.6%, 8.0% and 10.9% more time than the corresponding replica of VVWV, with 11, 21, 31 and 41 partitions, respectively, and low mobility. If, instead, we consider the extreme where only one replica is active, we substantially amplify such a difference to 24.5%, 59.3%, 64.2% and 56.6%, respectively.

Curiously, the relative advantage of VVWV over Deno decreases with replica order. We explain this by the fact that, in Deno, once multiple updates have committed at some replica (at the cost of one election per update), any other replica that does not know about the commitment of the multiple updates may learn of it in a single anti-entropy session with the former replica. Therefore, for the replicas that commit an update by collecting votes (typically the first ones to commit), Deno is not as efficient as VVWV; however, for the replicas that learn of commitment after the elections is complete (typically, the last replicas to commit), Deno can be as efficient as VVWV.

It is very relevant to note that, as we narrow update activity down to a single active replica, VVWV is the only protocol that exposes a graceful degradation of its AADs. In contrast, both Primary and Deno react to smaller numbers of active replicas with significant increases of their AADs. For example, with 31 partitions and low mobility, the third replicas of Primary and Deno to agree take, on average, 8.1% and 13.6% more time steps as we go from all to three active replicas. Their AADs further grow 49.3% and 48.0%, respectively, as active replicas drop from three to one. Regarding VVWV, the above variations are relatively small: 0% from five to three active replicas, and 12.9% from three to one active replicas.

Such an effect has important consequences to the relative performance, in terms

of AAD of the three protocols as less replicas are active. As we show previously, it boosts the relative performance of VVWV over Deno. Moreover, it causes VVWV to even outperform Primary in AAD for nearly all replicas, as we drop the number of active replicas to two or one. This observation is especially important when we take into account that numerous experimental evidence from real workloads of representative replicated systems suggest that the set of active replicas is typically very small in practice [Rat98].

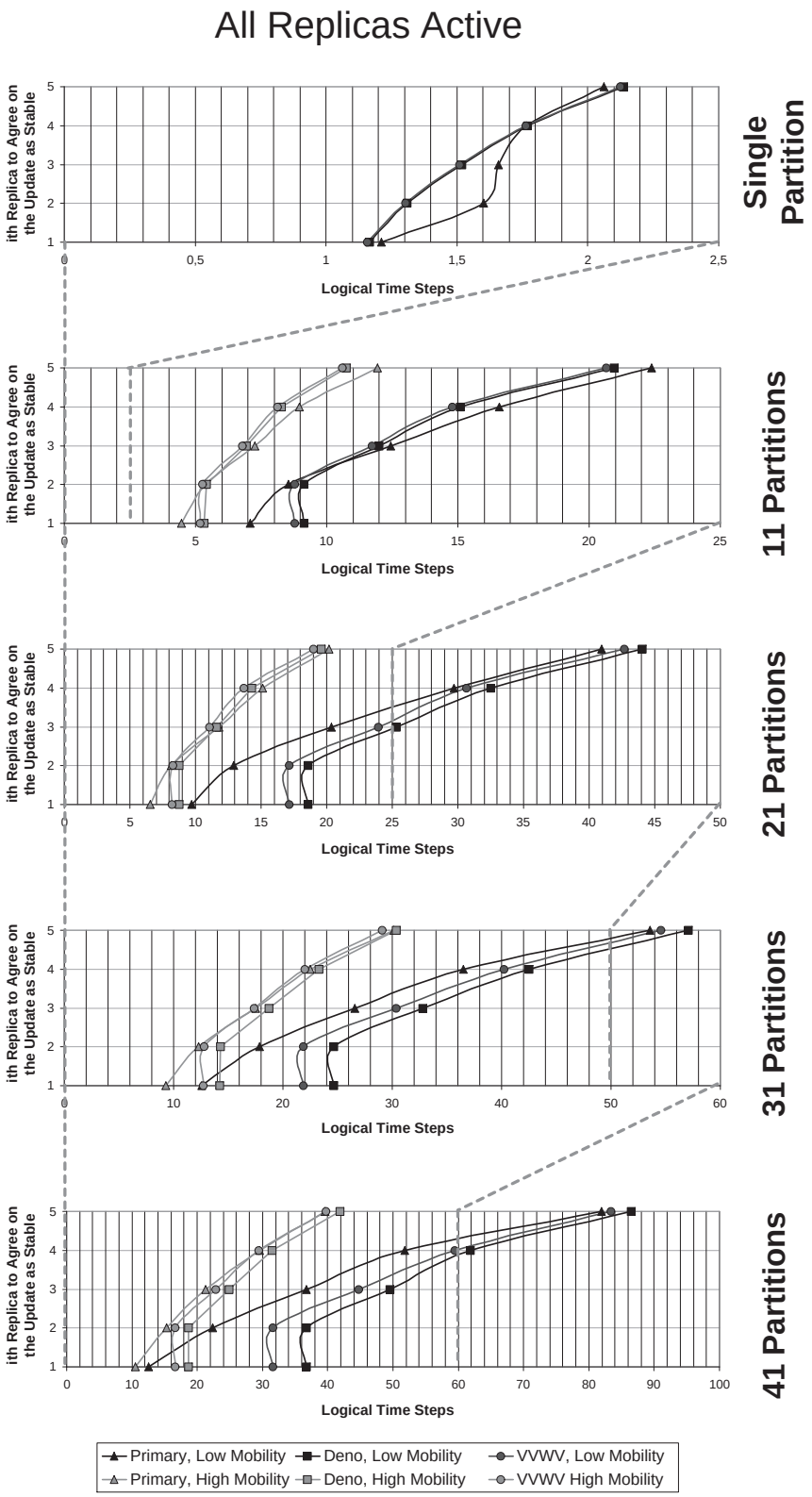


Figure 8.1: Average agreement delays vs. number of partitions, for low site mobility (*mobProb*=25%) and high site mobility (*mobProb*=75%). All replicas active, *updProb*=$\frac{0.5\%}{numActive} = 0.1\%$.

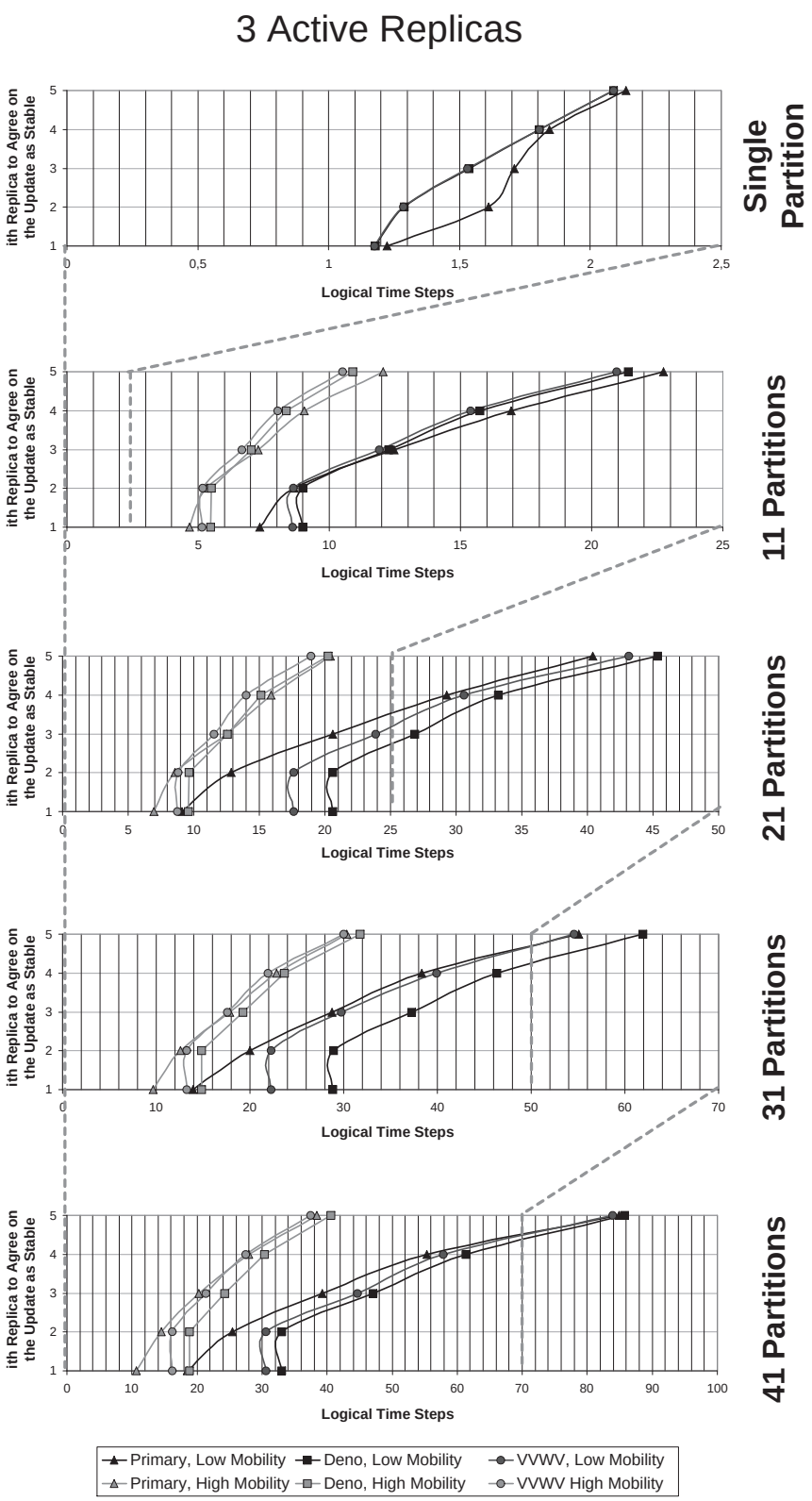


Figure 8.2: Average agreement delays vs. number of partitions, for low site mobility (*mobProb*=25%) and high site mobility (*mobProb*=75%). Three replicas active, *updProb*=$\frac{0.5\%}{numActive} = 0.167\%$.

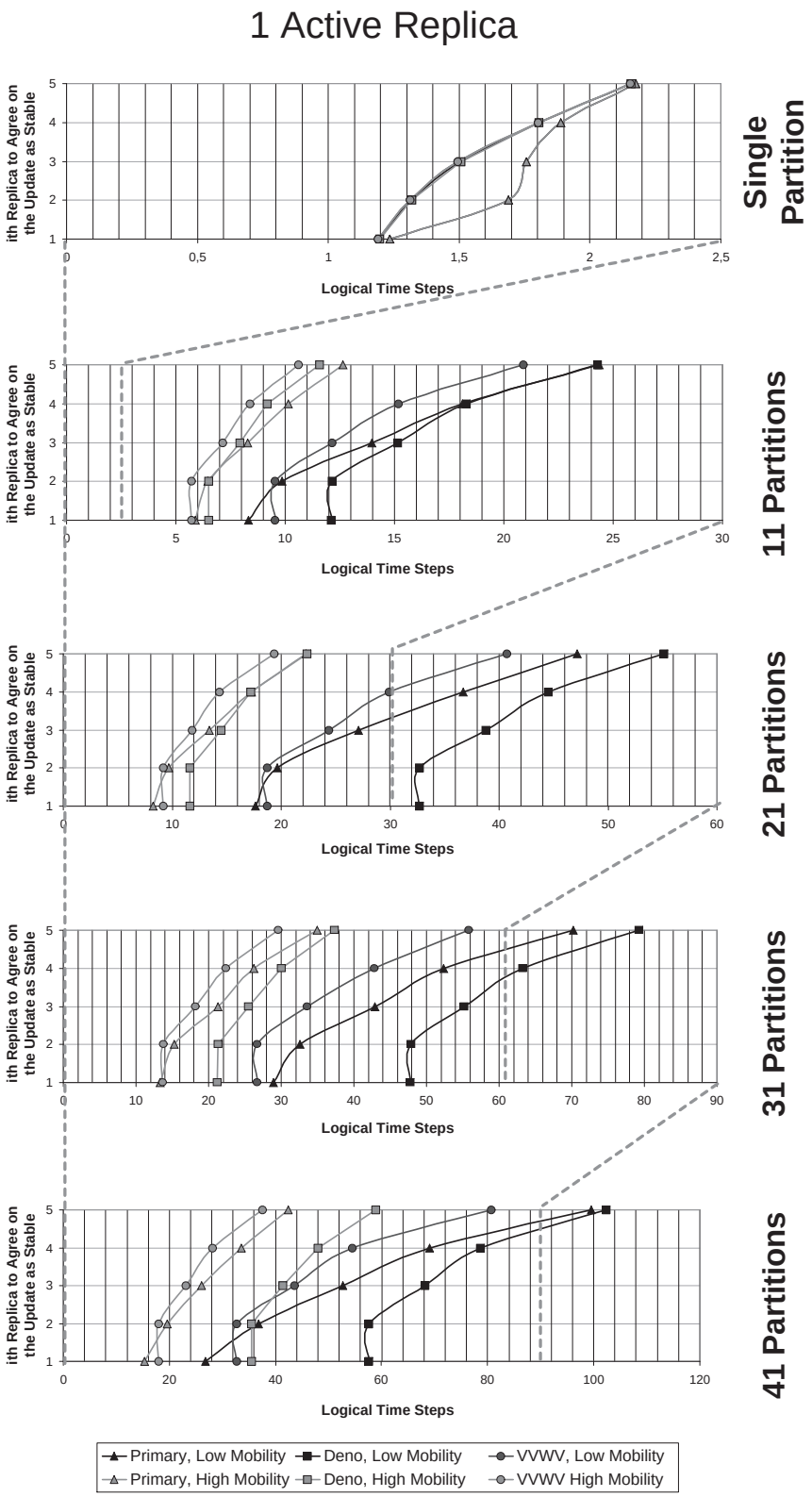


Figure 8.3: Average agreement delays vs. number of partitions, for low site mobility (*mobProb*=25%) and high site mobility (*mobProb*=75%). One replica active, *updProb*=$\frac{0.5\%}{numActive} = 0.5\%$.

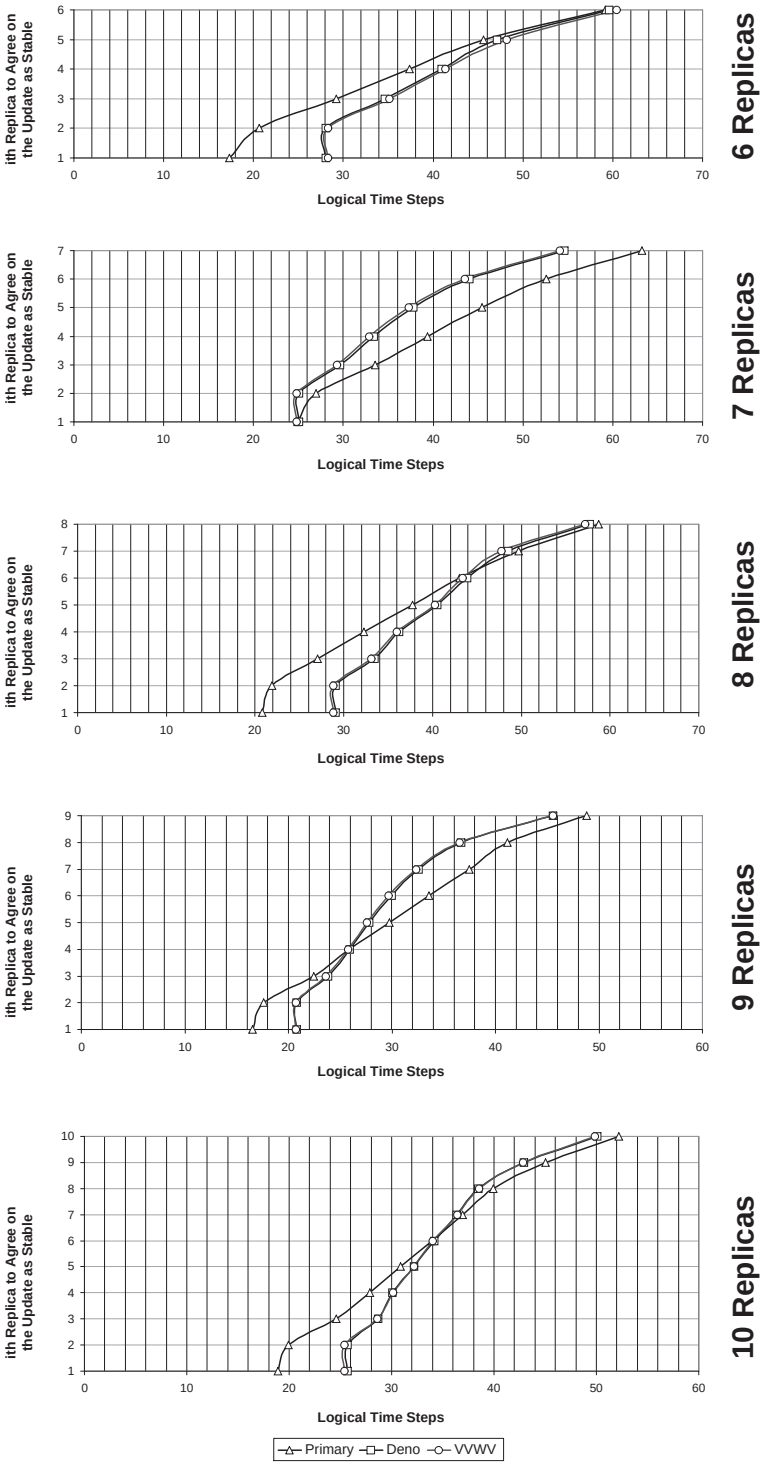


Figure 8.4: Average agreement delays vs. number of replicas. 31 Partitions, *mobProb*=25%, all replicas active, *updProb*=$\frac{0.5\%}{numActive} = 0.1\%$.

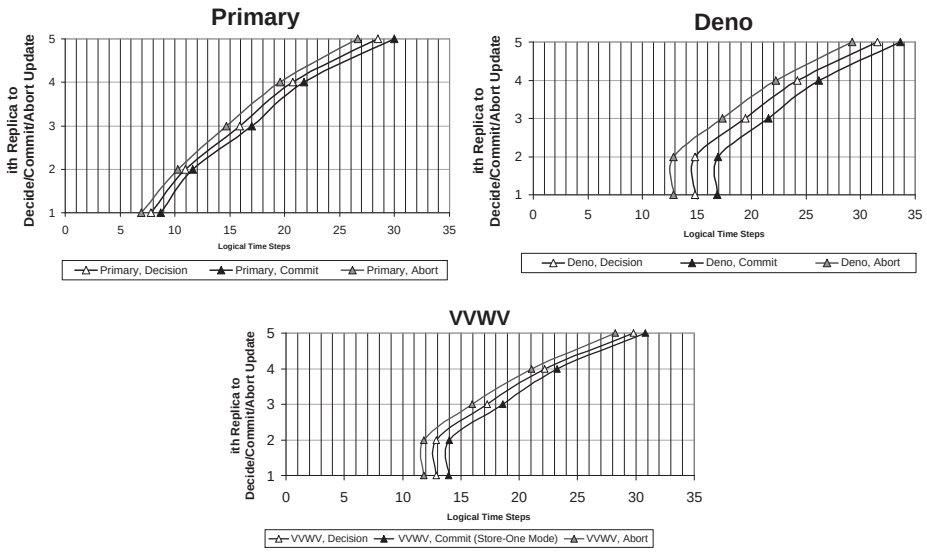


Figure 8.5: Average decision vs. commitment vs. abort delays in a high-contention scenario. 41 Partitions, *mobProb*=25%, all replicas active, *updProb*=$\frac{0.9\%}{numActive} = 0.18\%$.

8.1.1.2 Considerations on AAD

AAD gives a realistic picture of the user experience that a commitment protocol provides, since it tells us how much time a user has to wait before receiving either a commit or an abort decision on some tentative piece of work. However, we argue that AAD is not sufficiently meaningful as a measure of the effectiveness of a protocol. We support such an argument by decomposing AAD into commitment and abort delays, which allows for a better understanding of the actual meaning of the AAD measure.

Figure 8.5 distinguishes the commitment and abort delays (as well as the compound, agreement delays) of the three protocols in a particular setting, with 41 Partitions and *mobProb*=25%. We chose a relatively high update probability (*updProb*=0.18%), so that it induced a high contention scenario where comparable numbers of commitments and aborts occurred. As the results show, the time necessary to commit an update is markedly higher than to abort an update. In the setting that Figure 8.5 considers, commitment delay in Deno is 31% and 15% higher than abort delay for the first replica and last replica (to either commit or abort), respectively. Significant differences also exist with VVWV (18% and 9%) and Primary (25% and 12%). Moreover, other values of *mobProb*, *updProb*, *numPart* and *numActive* also yield relevant differences between commitment and abort delays.

Intuitively, we may justify the above difference by the fact that committing an update requires a complete election to initiate (by proposing the update as a candidate) and complete (by having the system vote for the update and electing it); whereas aborting does not. In fact, aborting a given update happens once some

update that is concurrent with the former commits; the election of the latter update may already have started when the aborted update was issued. As an example, consider that an update u_1, issued at time step 10, has already collected a majority of votes, but a given replica, r, has neither yet received the update nor its voting information. Assume that, at time step 21, r issues update u_2, concurrently with u_1. Then, at time step 22, r receives u_1 and its voting information. Hence, at time 22, r commits u_1, which has a commitment delay of $22-10=12$ at r; and, consequently, r aborts u_2, which has an abort delay of $22-21=1$ at r.

Therefore, the AAD measure tends to benefit (undesirable) protocols that may be slow to commit but abort frequently. When compared to a (more interesting) protocol that commits faster and more frequently, the former protocol may achieve AADs that are only slightly higher or possibly even lower than those of the latter protocol. This is due to the deceiving impact of *small delays of frequent aborts*, which replace longer-to-complete commitments. Furthermore, abort delays have a considerably high variance, due to their arbitrary nature, as described above. Hence, when comparing two protocols that commit similar number of updates, abort delay contributes with arbitrary noise to the overall AADs.

The previous observations reduce the value of comparing the effectiveness of the three protocols by their AADs. A more meaningful alternative is to consider only commitment delay, which we do in the next section.

#	ith Replica to Commit			ith Replica to Commit			ith Replica to Commit		
Partit.	1st	3rd	5th	1st	3rd	5th	1st	3rd	5th
1	-5,0%	-9,7%	2,7%	-3,9%	-10,4%	-2,2%	-3,6%	-14,9%	-0,9%
11	19,5%	-6,4%	-8,8%	9,9%	-8,0%	-11,2%	14,6%	-12,9%	-14,2%
21	54,7%	9,4%	0,4%	78,5%	11,6%	3,9%	5,8%	-10,0%	-13,7%
31	63,7%	10,7%	-0,6%	49,4%	3,0%	-1,4%	-7,7%	-21,7%	-20,5%
41	113,5%	17,9%	-1,9%	61,7%	11,0%	-5,4%	21,6%	-17,4%	-19,0%
	All Replicas Active			Three Active Replicas			One Active Replica		

Table 8.3: Relative increase in ACD from Primary to VVWV (i.e., $\frac{ACD_{VVWV} - ACD_{Primary}}{ACD_{Primary}}$), with low mobility, for different network settings.

8.1.2 Average Commitment Delays (ACD)

Figures 8.6 to 8.8 present average commitment delays (ACD) for each protocol, in different network scenarios, with varying numbers of partitions and replica mobility. As in the case of AAD, each figure considers a decreasing number of active replicas, therefore addressing different update models.

8.1.2.1 Analysis of ACD

Every observation that we make above for AAD remains valid in the case of ACD. Namely:

- ACD grows as partitions increase and as mobility decreases.
- Primary obtains lower ACDs for the first replicas, while the weighted voting protocols outperform Primary for higher order replicas to commit.
- VVWV attains lower ACDs than Deno, especially as the number of partitions grows.
- VVWV adapts better to narrower active replica sets, retaining comparable ACD levels, while Primary and Deno notice visible degradations in ACDs; as a result, the difference in ACD from VVWV to Deno grows substantially, and VVWV is able to completely outperform (i.e. for all replicas) Primary for a sufficiently low number of active replicas (in the case of our experiments, this occurs with a single active replica).

The essential difference between ACD and AAD, as we discussed in Section 8.1.1.2, is that ACD is immune to the noise that abort delays introduce in AAD. Since ACD does not take abort delays into account, it is generally higher than AAD, as expectable.

Tables 8.4 and 8.4 derive, from Figures 8.6 to 8.8, the relative variation in ACD from Primary to VVWV, and from VVWV to Deno, respectively.

Curiously, ACDs reveal that the noise of abort delays on AAD always benefits Primary over VVWV. In fact, the difference between the ACDs of Primary

#	ith Replica to Commit			ith Replica to Commit			ith Replica to Commit		
Partit.	1st	3rd	5th	1st	3rd	5th	1st	3rd	5th
1	0,9%	0,6%	0,5%	0,2%	0,4%	0,0%	0,5%	0,7%	0,1%
11	4,6%	3,1%	1,8%	4,6%	3,3%	2,5%	25,5%	21,8%	12,3%
21	14,7%	10,2%	4,5%	16,4%	12,2%	5,1%	78,7%	61,8%	38,7%
31	8,9%	4,3%	2,1%	28,1%	23,1%	12,8%	83,5%	67,0%	43,8%
41	31,4%	19,4%	10,4%	27,5%	20,7%	10,0%	87,6%	69,6%	32,5%
	All Replicas Active			Three Active Replicas			One Active Replica		

Table 8.4: Relative increase in ACD from VVWV to Deno (i.e., $\frac{ACD_{Deno}-ACD_{VVWV}}{ACD_{VVWV}}$), with low mobility, for different network settings.

and VVWV decreases in comparison to the corresponding difference in terms of AAD. For instance, to the highest difference between the AADs of VVWV and Primary, 152.6% (which occurs with 41 partitions, low mobility and all replicas active; see Table 8.1), corresponds an ACD of 113.5%. We verify the inverse when concerning Deno: the difference between the ACDs of VVWV and Deno increases substantially in comparison to the corresponding difference in terms of AAD. As an example, in the previous setting, the AAD of Deno is 16.3% higher than VVWV's, while the difference in ACD is of 31.4%. Summing up, the evaluation of the protocols from the point of view of ACD, rather than of AAD, is substantially more positive for VVWV.

We may further characterize each protocol by individually comparing the evolution from its AADs to its ACDs. Interestingly, in general (with some exceptions), Deno has the highest gap between AACs and ACDs; this is especially evident with lower numbers of active replicas. Since aborts tend to contribute to lowering AAD (in relation to ACD), such an observation suggests that Deno has lower commit ratios than the other protocols, particularly as the number of active replicas drops. Comparing VVWV with Primary, a similar phenomenon happens for a number of cases; accordingly, this suggests a lower commit ratio of VVWV in relation to Primary.

However, recalling the arbitrarity of the noise of abort delays in AAD, the above previsions are very limited in accuracy. The next section tries to validate them by studying the effective commit ratios of each protocol.

As Section 4.2.3 explains, the choice of the Store-All and Store-One storage modes of VVWV directly affects the commitment delay, thus ACD. All results that we present above were obtained with the Store-One mode. Therefore, in order to confirm that our conclusions generalize to the Store-All mode, we reran all the experiments described above using such a mode (using similar randomization seeds for the simulator, in order to reproduce similar executions). The measured differences are negligible. Figure 8.9 shows the results for one of the settings where ACD of both modes shows a larger difference. Overall, a maximum improvement of 1.8% was attained by storing the updates of all candidates (Store-All), which suggests that the more resource-efficient alternative of storing only the updates of

a replica's own candidate (Store-One) is acceptable.

8.1.2.2 Considerations on ACD

ACD gives a better understanding of the efficiency of a protocol, since it avoids the noise that abort delays introduce in AAD. However, it is still not a completely meaningful measure, due to the effect of hidden conflicts, as we explain now.

As Section 5.3 describes, aborts due to hidden conflicts are prone to occur since commitment is not immediate. By using a protocol that effectively commits faster, one reduces the probability of hidden conflicts; hence, on average, one aborts less updates. Therefore, the number of discarded updates also characterizes the effective efficiency of a protocol.

However, ACD does not take discarded updates into account. It is a useful measure to compare the efficiency of two protocols in executions where both commit a similar (or nearly similar) set of updates. However, when one protocol discards more updates because it is not as fast as another one, ACD does not necessarily expose such a difference. We ought, thus, to analyse the average commit ratios of Primary, Deno and VVWV in order to have a complete understanding of their commitment efficiency. This is the subject of the next section.

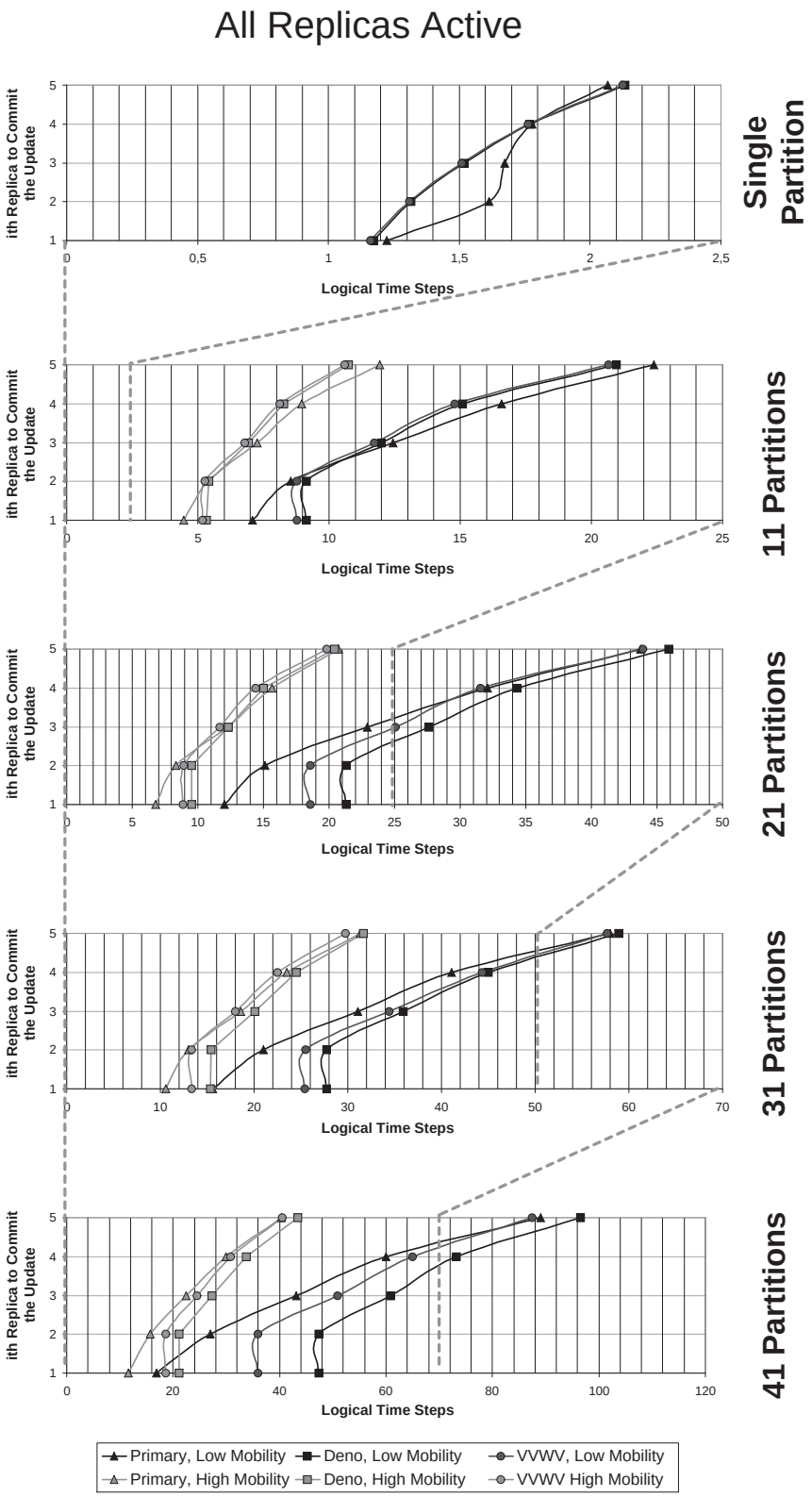


Figure 8.6: Average commitment delays vs. number of partitions, for low site mobility (*mobProb*=25%) and high site mobility (*mobProb*=75%). All replicas active, *updProb*=$\frac{0.5\%}{numActive} = 0.1\%$.

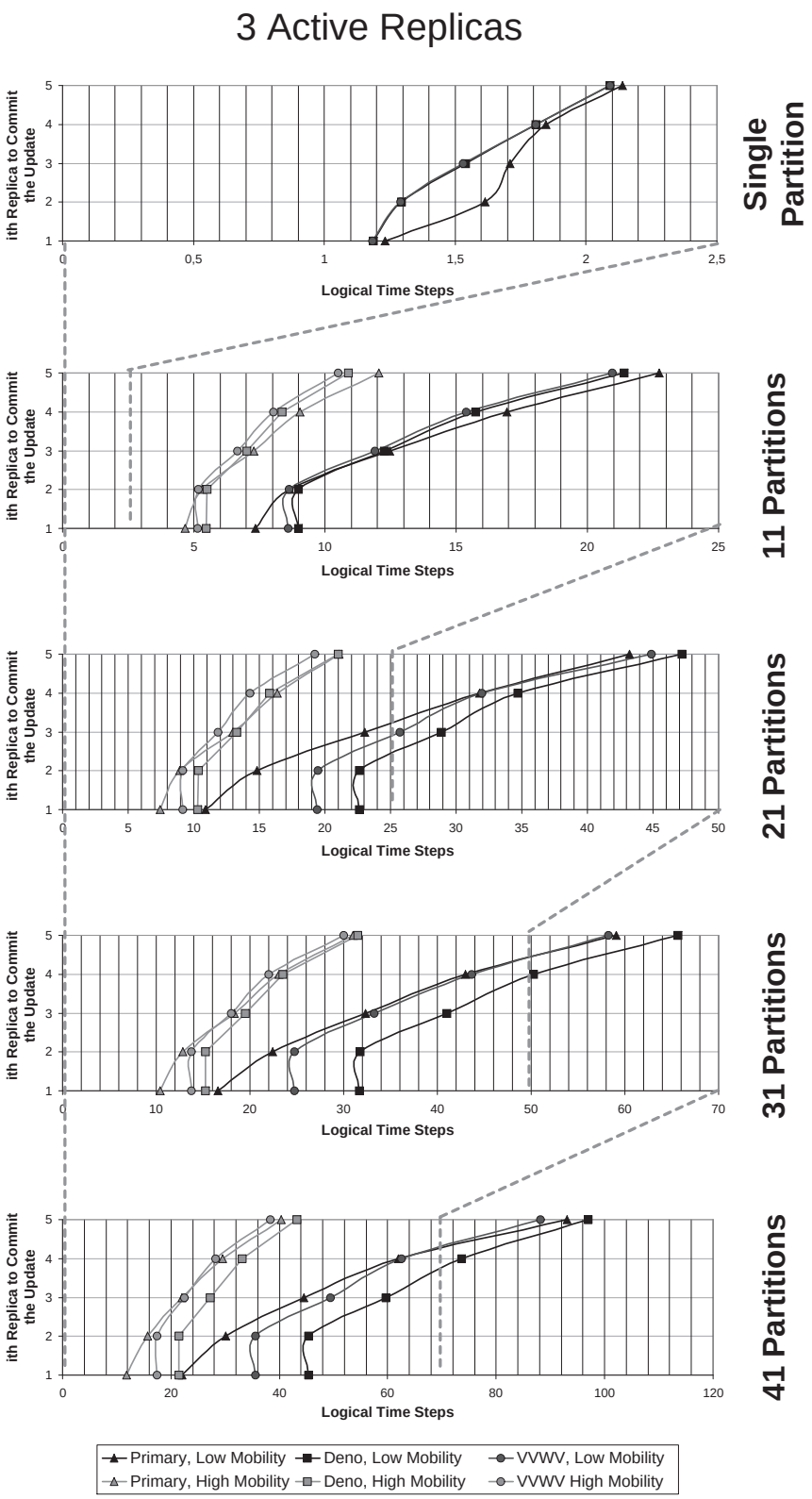


Figure 8.7: Average commitment delays vs. number of partitions, for low site mobility (*mobProb*=25%) and high site mobility (*mobProb*=75%). Three replicas active, *updProb*=$\frac{0.5\%}{numActive} = 0.167\%$.

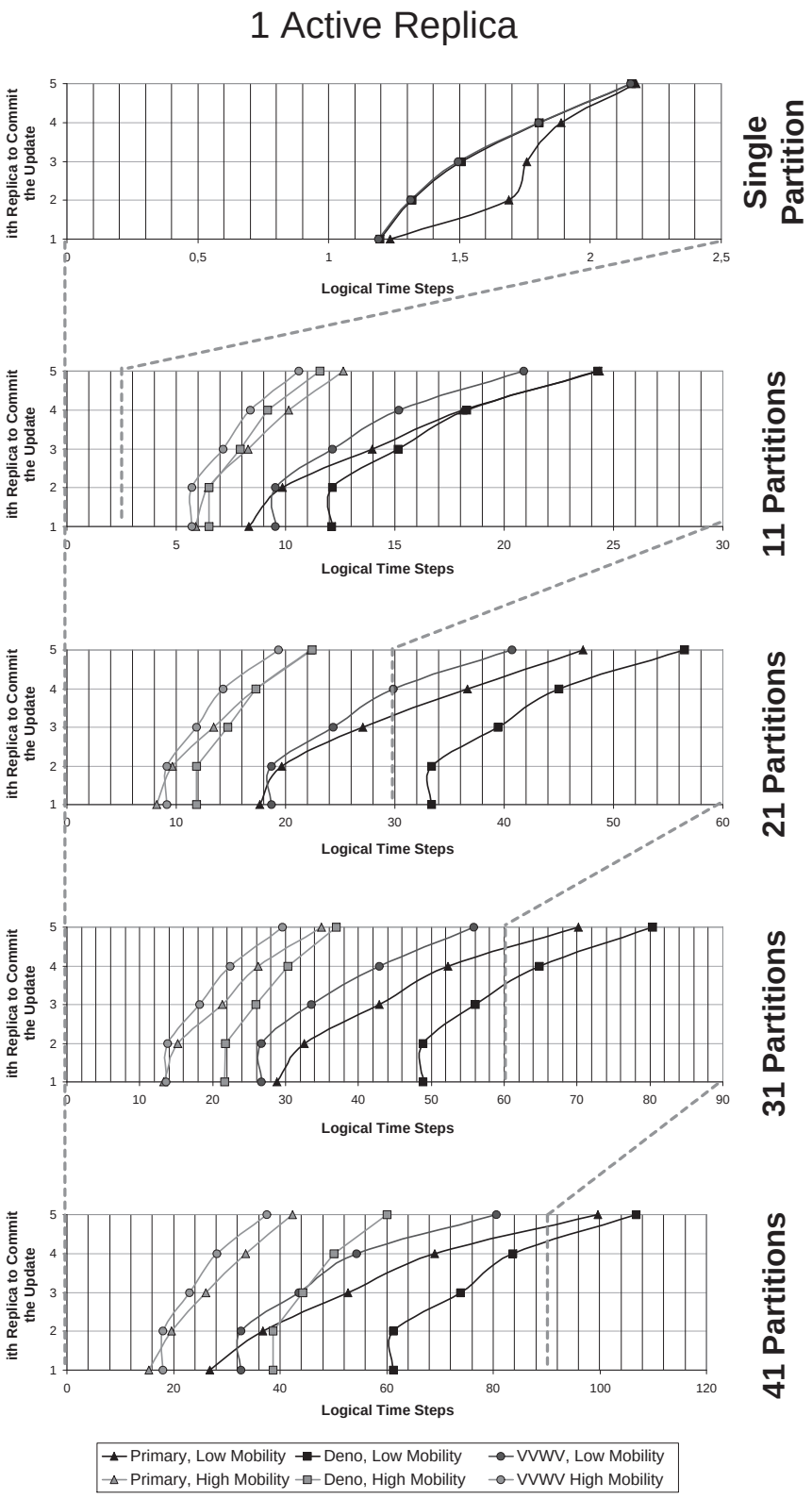


Figure 8.8: Average commitment delays vs. number of partitions, for low site mobility (*mobProb*=25%) and high site mobility (*mobProb*=75%). One replica active, *updProb*=$\frac{0.5\%}{numActive} = 0.5\%$.

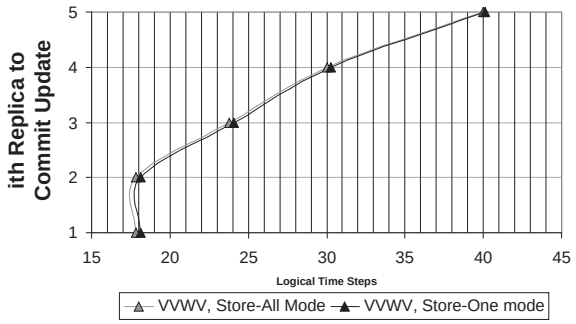


Figure 8.9: Commit delay of Store-All vs. Store-One modes of VVWV, in a high-contention scenario (41 Partitions, *mobProb*=25%, all replicas active, *updProb*=$\frac{0.9\%}{numActive} = 0.18\%$.). In Store-All mode, a site stores all concurrent tentative schedules that it receives by anti-entropy; once a commitment decision is made locally, the replica may immediately commit the corresponding updates. In Store-One, a site stores only the first tentative schedule that is receives; if the commitment decision chooses another concurrent schedule, the replica must wait for the corresponding updates to arrive in order to commit them.

8.1.3 Commitment Ratio

The previous study of AADs and ACDs provides hints on the relative frequency of updates that each protocol commits and aborts. This section measures the exact value of such frequencies, and validates some of the predictions that the previous study made.

We define commitment ratio as the percentage of issued updates that each protol commits as executed at all replicas (as opposed to the updates that it aborts). Figure 8.10 compares the commitment ratios of each protocol. As in previous measures, we consider both low and high mobility, and different numbers of active replicas, for and increasing number of partitions.

Commitment ratio is directly affected by the efficiency of each evaluated protocol, since if updates remain in their tentative state for longer periods, the probability of aborts is higher. The reason for this is hidden conflicts (see Section 5.3). Hence, lower commitment ratios reflect longer delays imposed by the update commitment process.

Figure 8.10 shows that, as expected, update commitment ratios decrease as the connectivity among replicas is weakened by an increasing number of partitions. However, Primary and VVWV are able to ensure higher ratios than Deno as partitioning grows and/or mobility decreases. Situations of multiple happens-before-related tentative updates occur more frequently as updates remain tentative for longer periods. Hence, such results are explained by the efficiency of the former protocols in commiting of multiple happens-before-related updates, in contrast to Deno. Such situations also increase as one considers a smaller number of active replicas. Accordingly, decreasing the number of active replicas accentuates the advantage of Primary and VVWV over Deno. On the other hand, Primary and VVWV have nearly similar ratios.

We should remark that higher update probabilities yielded equivalent, yet magnified, conclusions.

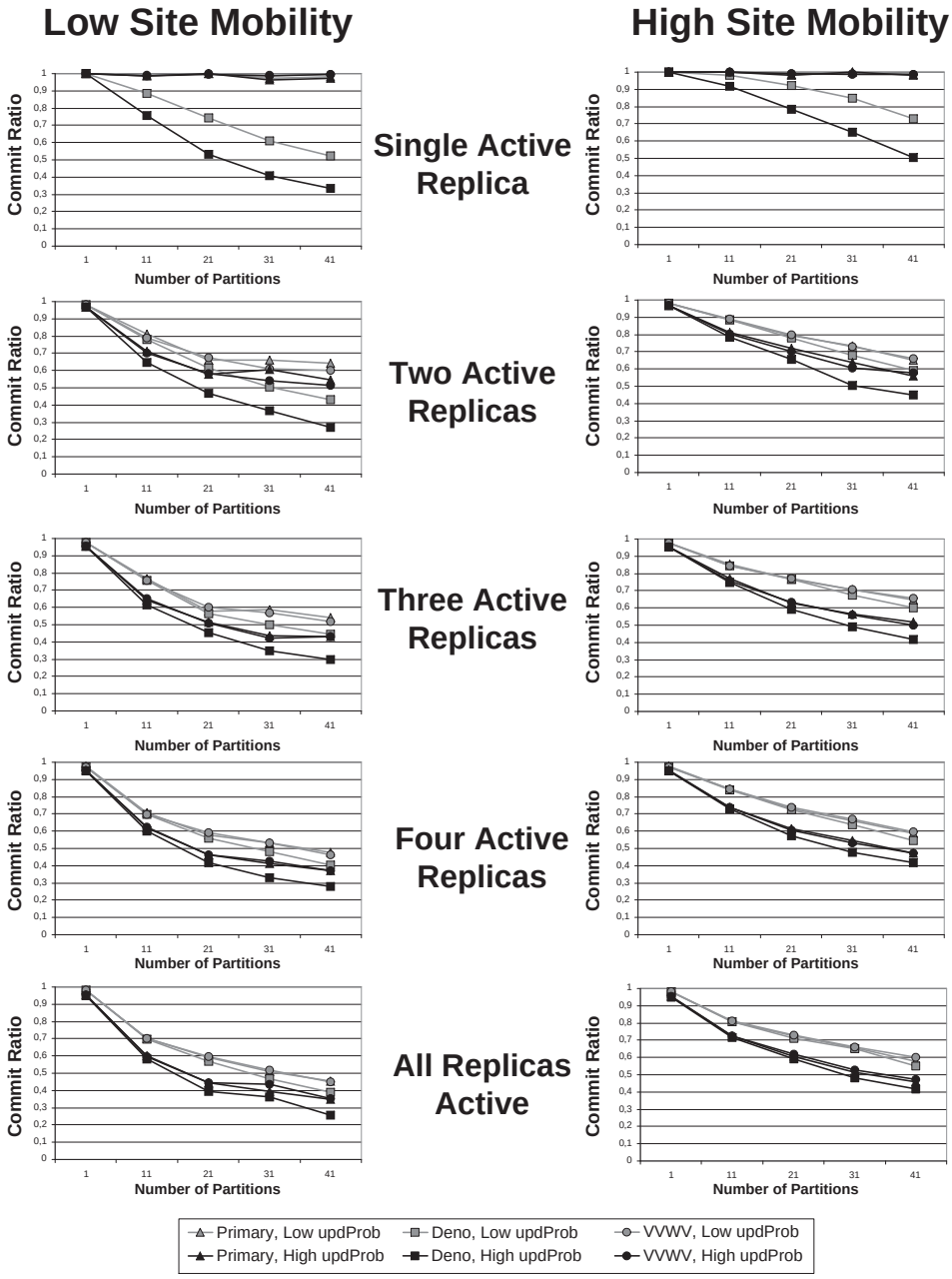


Figure 8.10: Commitment ratios vs. number of partitions for different numbers of active replicas. Both low and high site mobility are considered (resp., *mobProb*=25% and *mobProb*=75%); and both low and high update probabilities are considered (resp., *updProb*=$\frac{0.5\%}{numActive}$ and *updProb*=$\frac{0.9\%}{numActive}$).

8.2 Decoupled Update Commitment

We evaluate the exchange of consistency packets via non-replicas by immersing the system of five replicas, which we considered in the previous sections, in a larger system of 40 non-replicas. We then evaluated a decoupled protocol against an increasing portion of non-replicas that collaborate with the replicas by buffering and propagating consistency packets. In the remainder of the section, we designate each such collaborating non-replica an NRC.

For simplicity, this section limits its analysis to a single protocol, namely the decoupled version of VVWV (which Section 5.5 describes). As the remainder of the section shows, such a choice is representative enough to allow a meaningful analysis of every benefit that decoupled update commitment may achieve. We should also note that we do not evaluate buffer management issues. We assume that only a single object is replicated; hence, there is no contention of consistency packets of distinct objects at the buffers of NRCs. Such an evaluation is out of the scope of this thesis, and left for future work.

We evaluate the incorporation of NRCs against two variants. A first one assumes that no NRCs exist (*NoNRCs*); thus, the system behaves similarly as an isolated system of five replicas. In the second variant, NRCs carry enough information (namely, update data) to emulate complete off-line anti-entropy [PST+97, GRR+98, PSYC03] (*FullNRCs*).

We have simulated all variants in the same conditions and with the same randomization seeds. As in the previous experiments, each run lasted for 2000 logical time steps, and the results that we present next are average values from samples of five runs. Each run considered 45 nodes, among which five were replicas, all active. We reproduced each experiment with 0, 10, 20, 30 and 40 NRCs. Except when noted otherwise, the results presented below assume five partitions, and 20% mobility probability; the variation of such parameters was found not to invalidate our conclusions.

8.2.1 Agreement and Commitment Acceleration

A first experiment considered a contention-free situation, with *updProb*=0.02%[3] Figures 8.11 to 8.13 show how AAD varies in function of an increasing number of NRCs. As expected, AADs decrease as we introduce NRCs. This is the direct effect of NRCs. Such an effect is only visible, however, with more than 10 NRCs. For this reason, we omit the 10 NRCs case from the next results, as its null effect on AAD also implies a null effect on any indirect benefit resulting from agreement acceleration.

For numbers of NRCs higher than 10, AADs drop substantially. In general, weaker connectivity, either by more partitions and/or by lower mobility, accentuates the relative reduction in AADs that NRCs introduce. Interestingly, rela-

[3]This update probability was sufficiently low to not cause any case of concurrent updates during every experiment run.

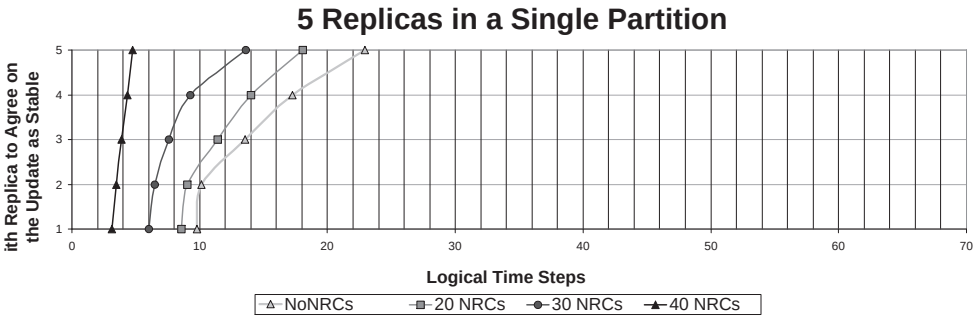


Figure 8.11: Average agreement delays, in a scenario of no update contention (*updprob*=0.2%), with five replicas in a single-partition network.

tive agreement acceleration is higher for higher order replicas. For example, with *mpbProb*=20% and five partitions, the first replica to agree takes 11.6%, 34.3% and 59.7% less time to agree with 20, 30 and 40 CNRs, respectively; such reductions become 12.7%, 35.7% and 65.8% (respectively) for the third replica to agree; and 15.5%, 37.3% and 73.8% (respectively) for the last replica to agree.

As expected from benefit A (commitment acceleration; see Section 5.3), not only do NRCs accelerate agreement, but also commitment. Figures 8.14 to 8.16 show that such an acceleration is substantial for a sufficiently high density of NRCs surrounding the replicated system. Reflecting the same tendency as with AAD acceleration, the relative acceleration in ACDs is higher for higher order replicas. As an example, five partitions and *mobProb*=20%, 40 NRCs reduce ACD by 46.5% and 16.0%, respectively for the first and last replicas to commit.

In practice, such an evidence means that commitment agreement is oftentimes the actual bottleneck delaying commitment, rather than update propagation. In other words, situations where replicas have received an update in a tentative form but have to wait for the corresponding commitment agreement before committing the update do occur in significant frequency.

One perhaps surprising fact becomes evident when we turn our attention toward the difference between incorporating NRCs and FullNRCs. For a sufficient number of NRCs, their ACD acceleration is comparable to, or better than, the one that FullNRCs achieve (though with a lower number of FullNRCs). For example, with five partitions and *mobProb*=20%, 30 NRCs ensure better commitment delays than 20 FullNRCs for all but the last replica to commit.

This suggests that the commitment acceleration that a set of resource-rich FullNRCs attains may still be achieved by replacing them with a larger set of resource-constrained NRCs. This constitutes a very significant observation, which is especially meaningful in the context of ubiquitous computing and sensor networks, where the latter (resource-constrained NRCs) are the norm.

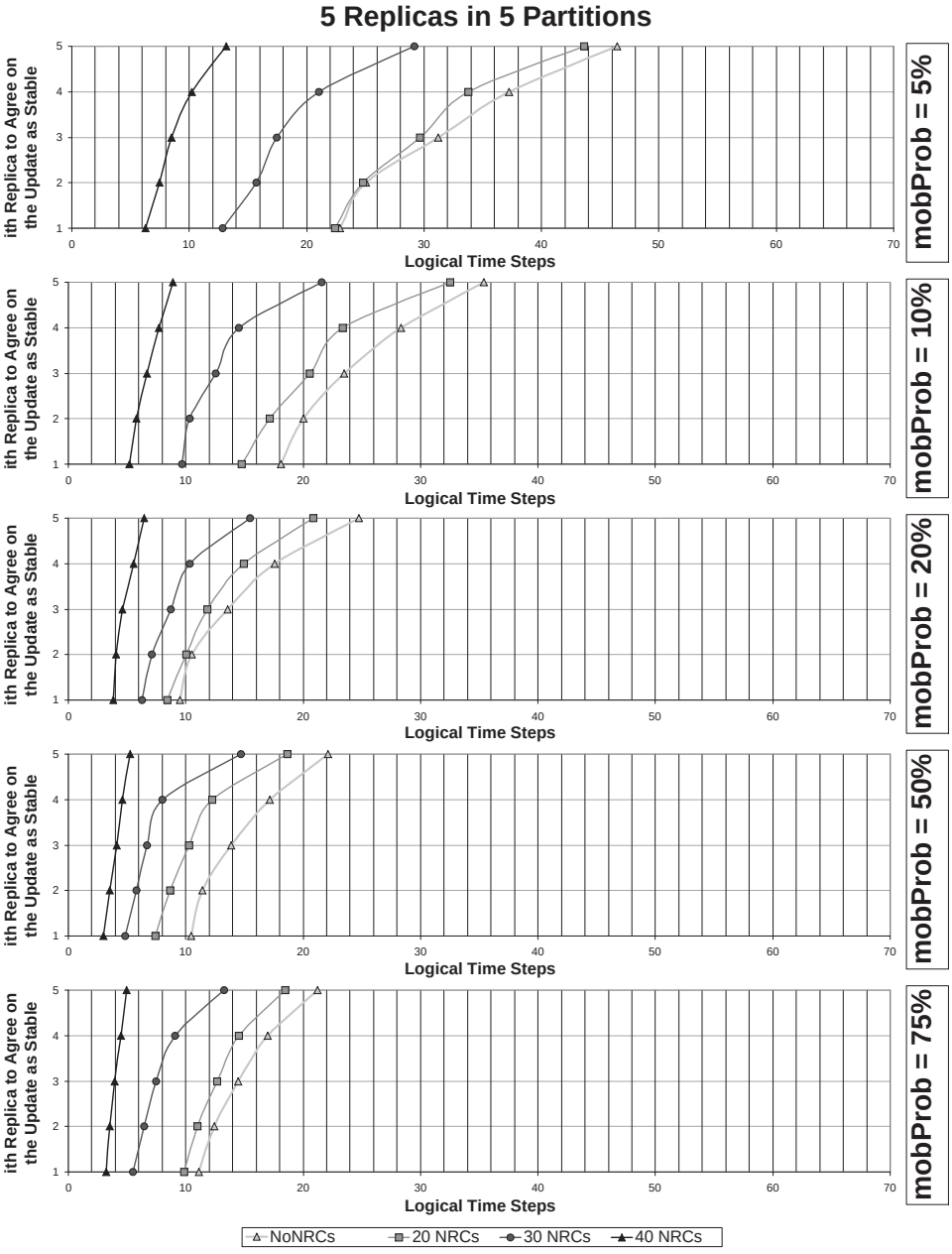


Figure 8.12: Average agreement delays, in a scenario of no update contention (*updprob*=0.2%), with five replicas in a 5-partition network.

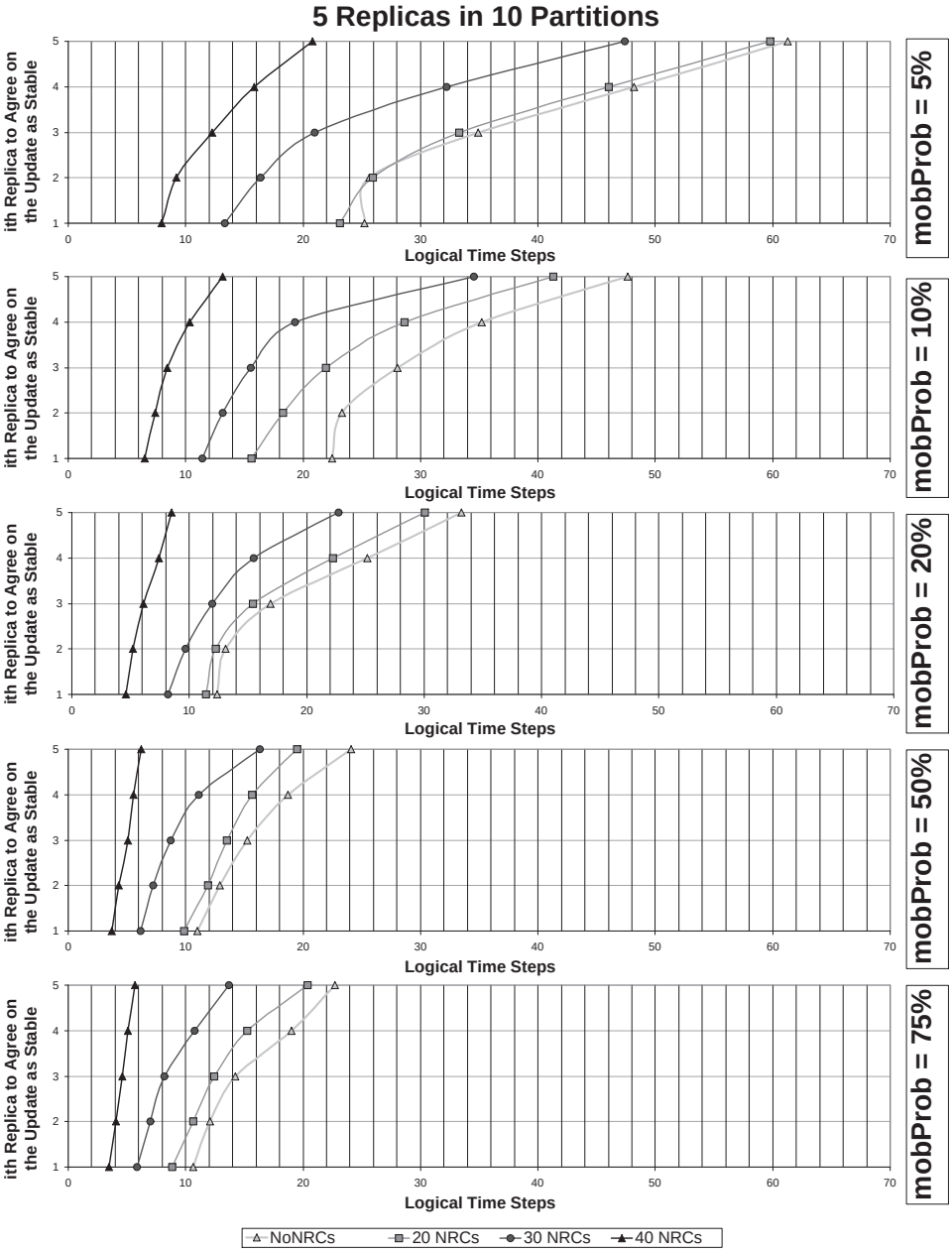


Figure 8.13: Average agreement delays, in a scenario of no update contention (*updprob*=0.2%), with five replicas in a 10-partition network.

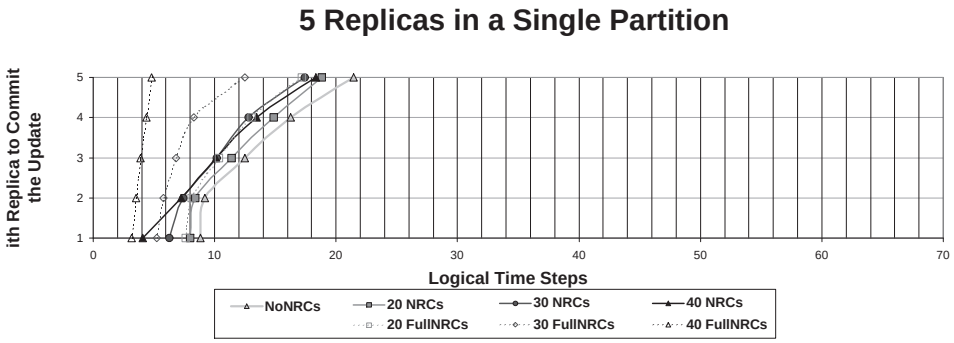


Figure 8.14: Average commitment delays, in a scenario of no update contention (*updprob*=0.2%), with five replicas in a single-partition network.

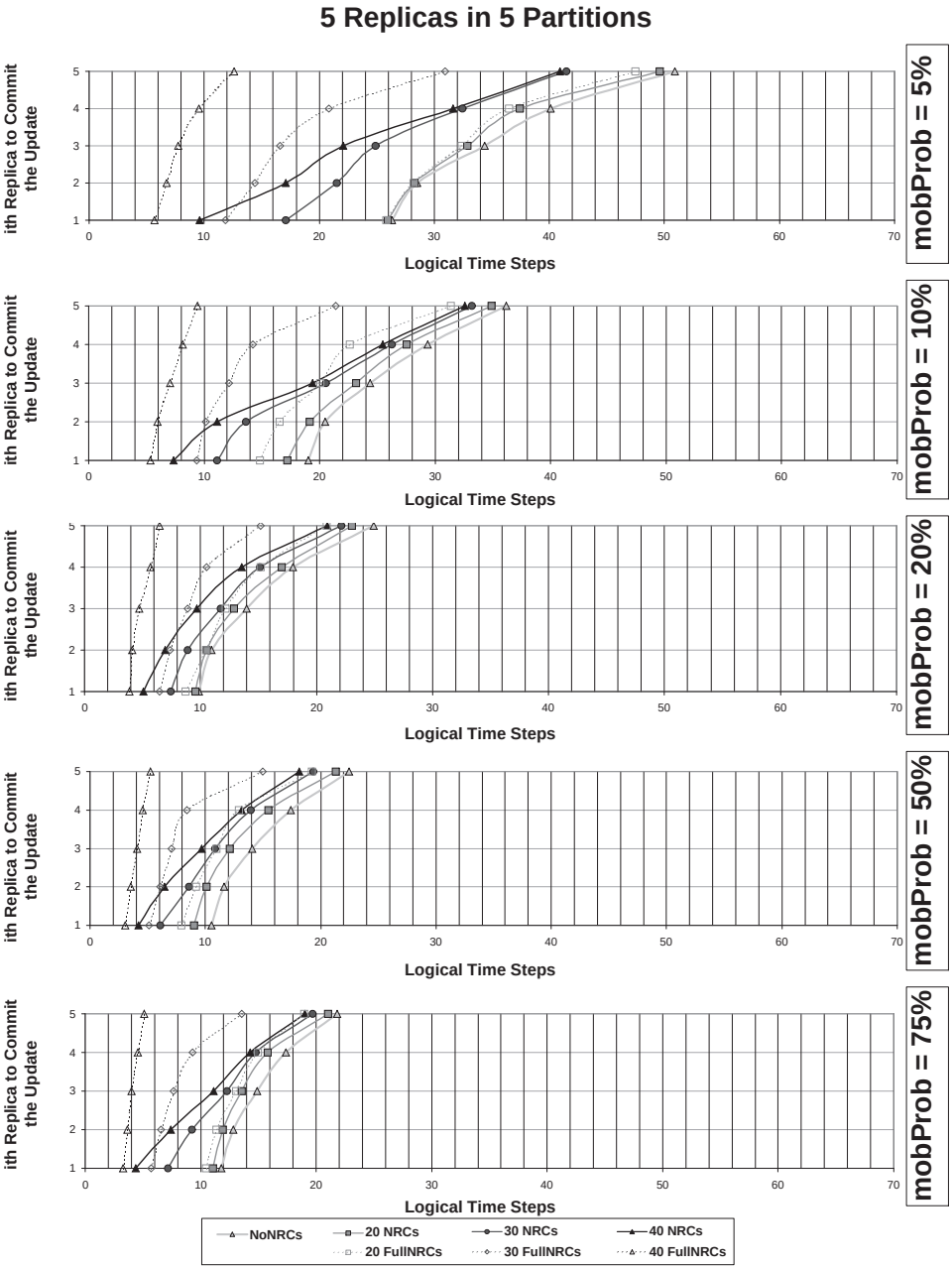


Figure 8.15: Average commitment delays, in a scenario of no update contention (*updprob*=0.2%), with five replicas in a 5-partition network.

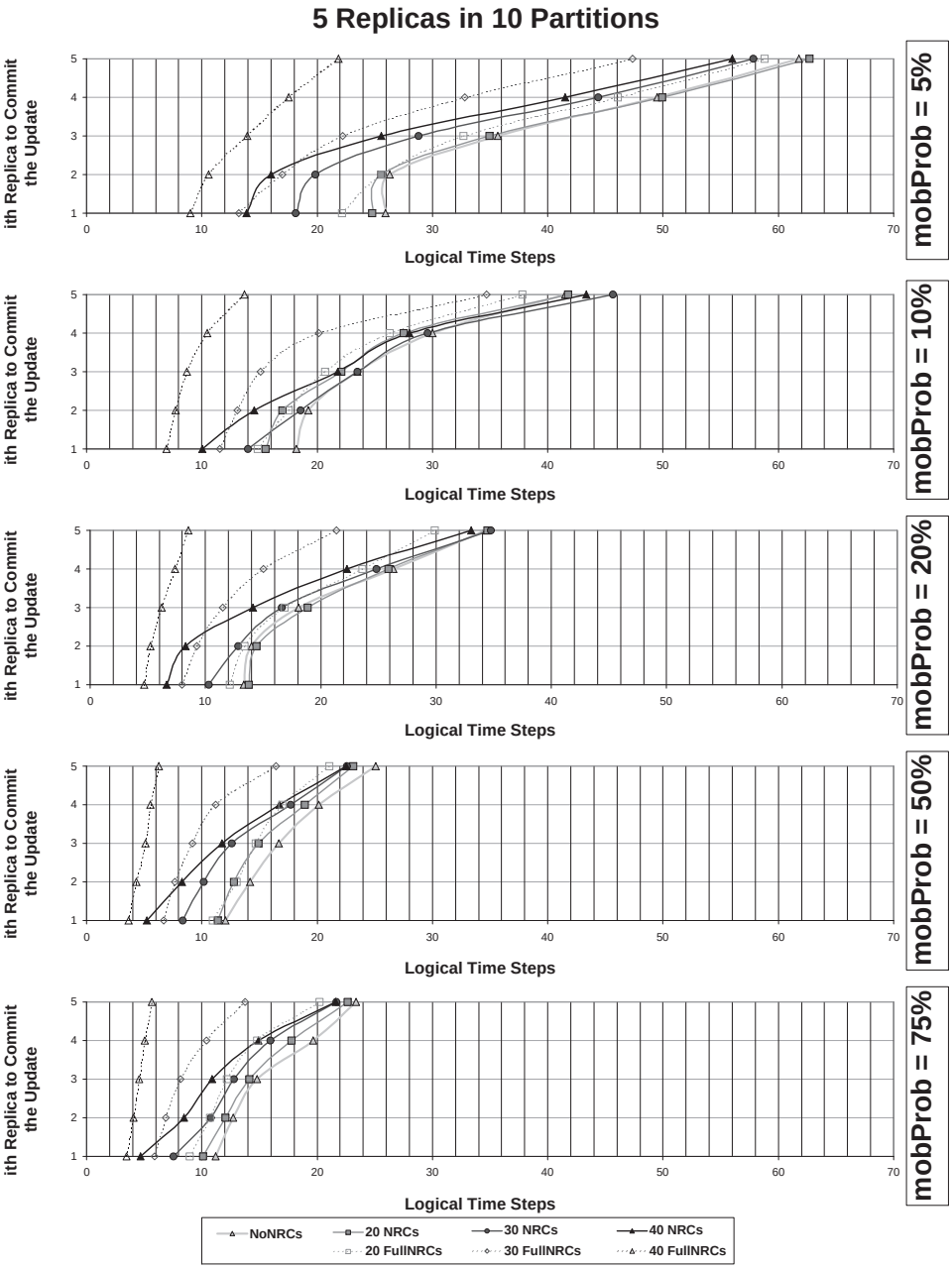


Figure 8.16: Average commitment delays, in a scenario of no update contention (*updprob*=0.2%), with five replicas in a 10-partition network.

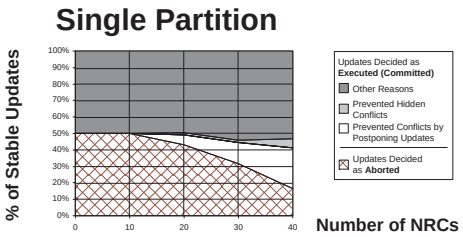


Figure 8.17: Commit and abort decision ratios, with detailed analysis of the reasons behind the aborts prevented with CNRs, in comparison to a NoCNR setting. One partition, all replicas active, high update contention (*updprob*=2%).

8.2.2 Abort Minimization

A second experiment considered a scenario of high update contention, yielding frequent conflicts (*updprob* of 2%). This allowed us to study the impact of CNRs on abort minimization; namely, to measure benefits B (*conflict prevention*; see Section 5.3) and C (*reduction of hidden conflicts*; see Section 5.3). We exploited benefit B as follows: when simulation step 3 (update issuing) decides to generate an update at some replica *r*, *r* defers the update's generation until the schedule corresponding to $m\text{-}sch_r$ becomes available.

More than determining how many update aborts we avoid by introducing CNRs into the system, we are interested in distinguishing the reasons behind each saved update. For such a purpose, we have instrumented the simulator in order to trace, from the updates that commit in a run with CNRs, which ones abort in a run with similar parameters and randomization seeds but with no CNRs. For each such update, the simulator further determines, when possible, the reason behind the abort prevention; namely, whether:

- the update aborts in the NoCNRs run due to a hidden conflict;
- the update aborts in the NoCNRs run and has its generation postponed in the CNRs run (where it commits).

For most prevented aborts, we were able to correlate them to benefits B or C. In some cases, however, neither one of the above conditions holds. In such cases, we leave the reason for the abort prevention unspecified.

Figures 8.17 and 8.18 present the ratios of committed and aborted updates, for varying numbers of partitions, mobility, as we incorporate more CNRs into the system. Starting at 20 NRCs, we observe important decreases on the frequency of aborted updates. For example, with *mobProb*=20%, and five partitions, 20, 30 and 40 CNRs cause aborted updates to drop by 8%, 40%, 73%, respectively.

More valuable conclusions are possible by analyzing the reasons behind such improvements in abort ratios. Starting by benefit B, it contributes to a substantial abort prevention by suggesting users and applications not to issue updates during

conflict-prone periods, when consistency meta-data indicates concurrent work going on. Its impact becomes dominant as the number of NRCs grows. For instance, with five partitions and *mobProb*=20%, 5.4%, 10.3% and 27.5% of the issued updates commit because their generation was postponed thanks to the meta-data disseminated by 20, 30 and 40 CNRs, respectively; had they not been postponed, they would have aborted.

Of course, such an improvement comes at the cost of temporary (though optional) unavailability. It is, thus, important, to determine the extent of conflict-prone periods, where users and applications are suggested not to issue updates. Figure 8.19 shows the duration of such periods, relatively to the total duration of each simulation run. Additionally, it depicts how many updates, initially determined to be issued inside one such period, were postponed. Its results show that the percentage of conflict-prone logical time steps is substantial, growing with the number of CNRs.

However, when compared to the percentage of aborts that it prevents, conflict-prone periods prove to be reasonably effective. We illustrate such a statement with an example with the same settings as above (five partitions and *mobProb*=20%). In this case, the consistency packets disseminated by 20, 30 and 40 CNRs yield, on average, 8.2%, 20.7% and 42,6% of conflict-prone time steps, respectively; during such periods, an average of 15.2, 41.6 and 83.0 updates were postponed, respectively. By comparing with the previous abort prevention results, we deduce that 66.0%, 49.8% and 64.6% (with, respectively, 20, 30 and 40 CNRs) were decisive, since the corresponding updates would abort commit if issued earlier. Such percentages assert the high value of the meta-data carried by CNRs as hints to which are the safest periods to generate updates, in terms of conflict probability.

Returning our attention to Figures 8.17 and 8.18, we also observe that benefit C has a relevant contribution to abort minimization, by preventing hidden conflicts. Taking the case of five partitions and *mobProb*=20% as a representative example, 20, 30 and 40 NRCs prevent 1.0%, 2.4% and 3.4% of issued updates from aborting, respectively. From the simulator traces, we know that the NoNRCs runs had an average of 16% of aborts due to hidden conflicts. Therefore, NRCs largely prevents them (30 and 40 NRCs avoid more than 50% of such hidden conflicts).

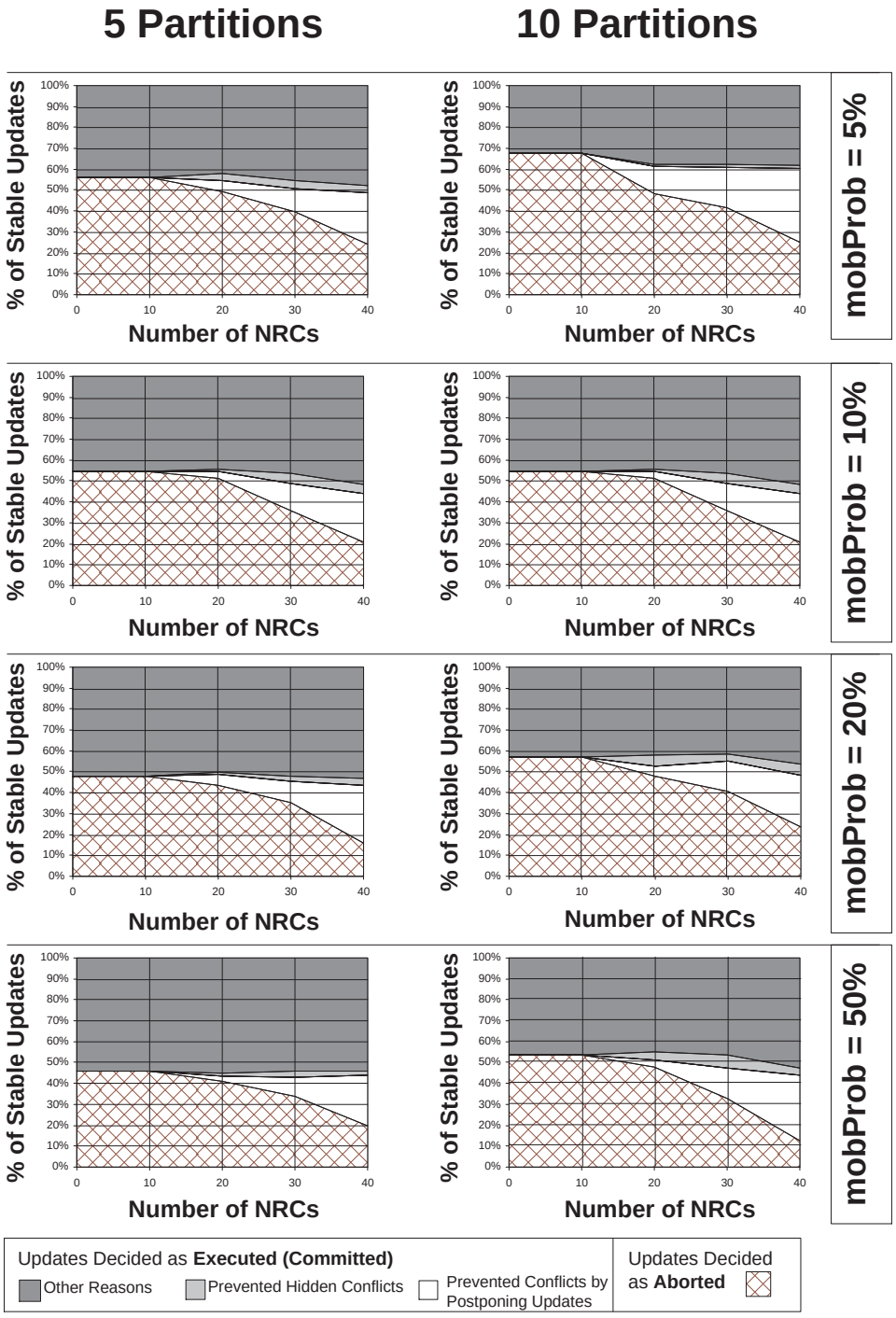


Figure 8.18: Commit and abort decision ratios, with detailed analysis of the reasons behind the aborts prevented with CNRs, in comparison to a NoCNR setting. Five and ten partitions, all replicas active, high update contention (*updprob*=2%).

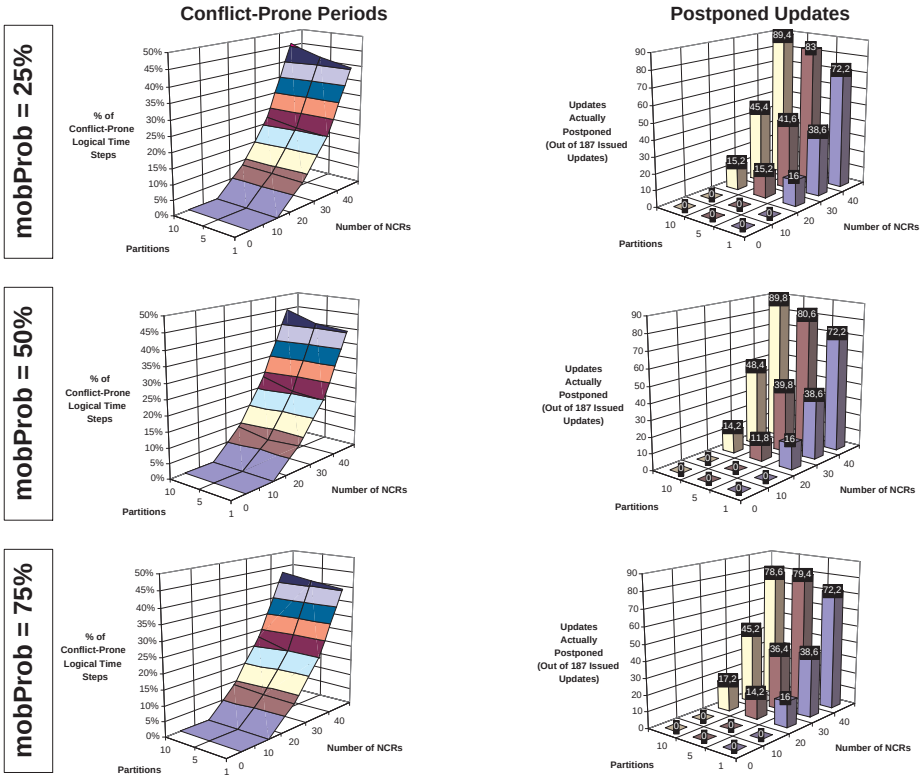


Figure 8.19: Update postponement statistics.

8.2.3 Improvements on Update Propagation Efficiency

The benefit of commitment acceleration arises from situations where, without any CNRs, agreement (regarding some update) occurs after the reception of the corresponding update. As we have shown, CNRs accelerate agreement, possibly anticipating it to a point in time that is closer to the moment where the replica receives the update (in a tentative form). Agreement acceleration may, however, go beyond the above limit, causing agreement to occur even *before* the reception of the corresponding update. In this case, the replica will already receive the corresponding update in a stable form, rather than in a tentative form. Section 5.3 designates such situations as *pre-reception agreement*. As it explains, the protocol may profit from such pre-reception agreement to more efficiently propagate the corresponding updates. Namely, the protocol needs not propagate updates that are already decided to abort; and it may more efficiently propagate updates that are already decided to commit (see Section 5.3).

To understand how often pre-reception agreement occurs due to the incorporation of CNRs, we have counted how many received updates were still in a tentative form, versus how many were already decided for commitment at the receiving replica. For the former, we further distinguish between the updates that the protocol later decided to commit, and to abort. In order to observe the three situations, we have considered the same update probability as in the previous section (*updprob*=2%), which yields frequent concurrent updates.[4] Such a setting yields frequent concurrent updates.

Figure 8.20 presents the number of updates that each replica individually received, distinguishing the updates by their form (already stable, then committed; tentative, eventually committed; or tentative, eventually aborted) at propagation time; for an increasing number of partitions.

An immediate observation is that the number of update commitments (i.e. resulting from the updates propagated as stable, then committed, plus the updates propagated as tentative, and eventually committed) grows with the number of CNRs. This is a direct consequence of the effect of CNRs on abort minimization, which the previous section studied. From the network efficiency point of view, an ideal protocol would: (i) propagate only updates that eventually commit (i.e. it would not unnecessarily propagate any update that would eventuallly abort); and (ii) propagate every update only once pre-reception agreement (concerning such an update) occurs.

Increasing the density of CNRs pushes the protocol toward such an objective. On the one hand, increasing CNRs significantly reduces the relative network overhead associated with (unnecessarily) propagating updates that the receiving replicas later abort. From the results in Figure 8.20, we obtain such a relative overhead by dividing the number of propagated updates that eventually abort by the num-

[4] We should add that we obtained similar conclusions in a contention-free setting (naturally, in this case, all tentative updates later became decided as committed).

ber of propagated updates that eventually commit.[5] The relative overheads are as follows:

NumPartitions	NoCNRs	20 CNRs	30 CNRs	40 CNRs
1	36.7%	24.4%	9.6%	2.4%
5	35.5%	30.4%	18.8%	3.0%
10	31.3%	28.4%	12.6%	2.0%

Table 8.5: Relative overhead with unnecessarily propagated updates (which the protocol later aborted).

Especially for 30 and more CNRs, the attained acceleration in AAD drastically lowers the relative overhead associated with unnecessarily propagated updates.

On the other hand, CNRs also increase the number of updates that each replica already receives in a stable (for commitment) form, which may thus be propagated more efficiently. The following table presents their proportion within the total number of updates that the protocol propagates and eventually commits, using the results in Figure 8.20:

NumPartitions	NoCNRs	20 CNRs	30 CNRs	40 CNRs
1	33.9%	47.3%	63.7%	80.3%
5	35.5%	45.1%	61.4%	76.1%
10	40.3%	47.0%	59.5%	73.2%

Table 8.6: Proportion of eventually-committed updates that the protocol already propagates in a stable form.

These results show that the impact of CNRs on the proportion of updates that the protocol propagates in a stable form is significant. Notably, such an increase is more marked when less partitions exist. We know, from Section 5.3 (benefit D), that a higher proportion of updates that are stable at propagation time allow more efficient update propagation. The exact quantification of such an improvement is application-specific, hence not obtainable from our generic measures and out of the scope of our evaluation.

[5] Taking a look at Figure 8.20, one might try to assert the magnitude of the increase of pre-reception agreement situations by comparing the absolute reductions in the number of updates that the protocol propagates and later aborts. However, such a comparison does not fairly measure the impact of increased frequency of pre-reception agreement. In fact, the absolute reduction in propagated updates that later abort is rather a combined result of (i) abort minimization, which the previous section addressed, and (ii) more frequent situations of pre-reception agreement.

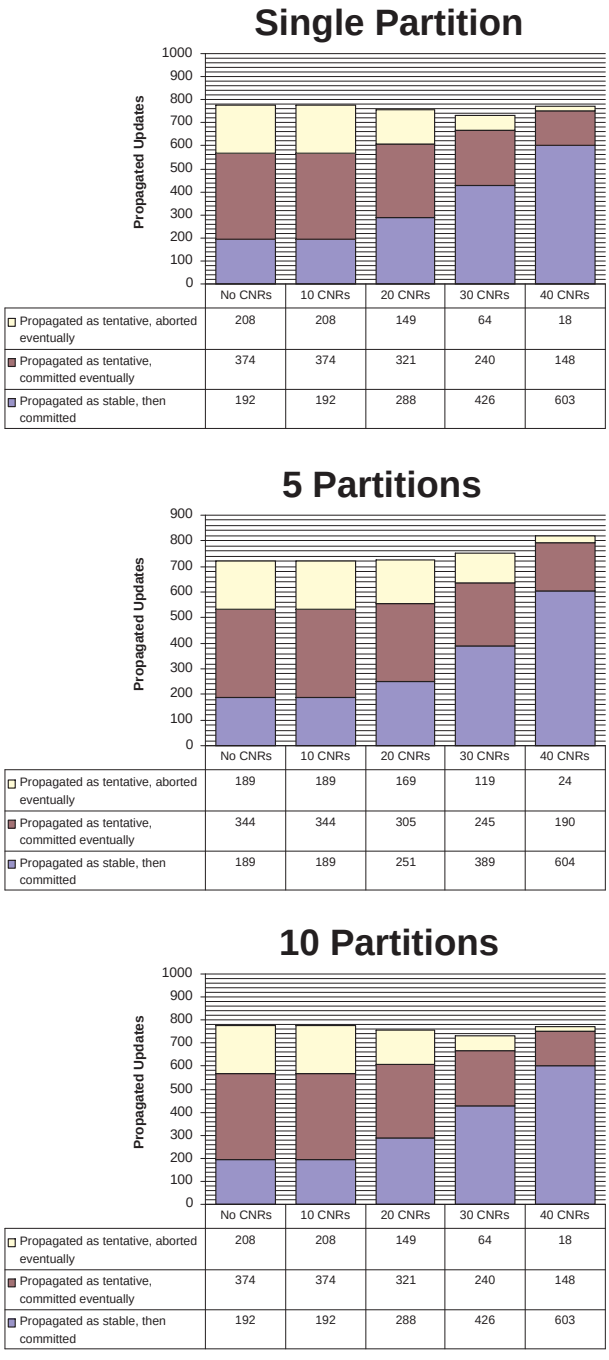


Figure 8.20: Frequency of propagated updates, distinguished by their form, with *mobProb*=20%.

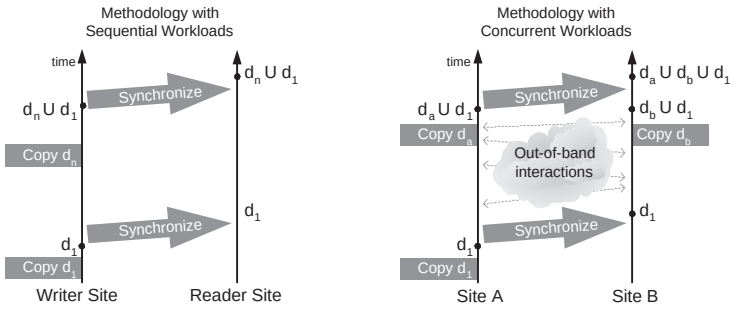


Figure 8.21: Temporal diagrams of experimental procedures followed for each type of workload, sequential and concurrent.

8.3 Versioning-Based Deduplication

This section evaluates dedupFS under different aspects, from bandwidth efficiency to synchronization performance, comparing it with relevant alternatives such as rsync, xdelta, diff, LBFS and gzip-compressed file transference. We analyze experimental results taken from representative workloads.

8.3.1 Experimental Setting

We have obtained the results by synchronizing different workloads between two distributed sites.

A first site ran Ubuntu 6.06 (kernel version 2.6.15) on a Pentium 4 3GHz processor with 1GB of RAM. The second site ran Debian 4.0 (kernel version 2.6.18) on an Athlon 64 processor 3200+ with 2 GB of RAM. All experiments were run during periods of negligible load from other processes running at each machine (approximately $<$ 1% CPU usage by other applications). A 100 Mbps full-duplex Ethernet LAN interconnected both sites.

8.3.1.1 Methodology

We consider different, real workloads when emulating collaborative update activity at each site. For different workloads and settings, we measured both the transmitted volume of data, and the access and synchronization performance of each evaluated solution. For the first measures, we use statistical data that each evaluated solution outputs (as we explain next). Hereafter, we use the symbol B to represent an 8-bit byte, and the symbols KB, MB and GB to, respectively denote 1024 B, 1024 KB and 1024 MB.

With respect to performance measures, we used the `time` command of Linux, with millisecond precision. Unless when stated, the performance results are an average of 3 executions of the same experiment.

Each workload represents the evolution of a shared directory, containing a given set of files and sub-directories, along a given time period. The workload directory consists of a single r-unit. We study two types of workloads: *sequential* and *concurrent* workloads.

Sequential workloads. Sequential workloads capture scenarios where a single writer site is producing a chronologically ordered sequence of versions of the workload directory, $d_1, d_2, .., d_n$. At each version, the writer site may modify files and sub-directories from the previous version, as well as create new files and sub-directories. Reader sites, in turn, keep replicas of the workload directory and occasionally synchronize with the writer's replicas, in order to obtain new versions as they become available.

Except where noted, we restrict our evaluation to the transition from d_1 to d_n, for two reasons. The first reason is simplicity, since analyzing each intermediate version transition is not relevant for most of following evaluation.[6] Secondly, for most workloads we consider (e.g. open-source code projects), the intermediate versions are not publicly available.

The experimental procedure emulates the synchronization of the shared workload directory from one writer site to one reader site.[7] Starting with both sites empty, the procedure consists of the following steps, which Figure 8.21 depicts:

1. We recursively copy the contents of d_1 into the writer site. In the case of dedupFS, such files and directories are copied into directory "1" of the local dedupFS mount directory.

2. We synchronize the (previously empty) reader site with the writer site. In this moment, both sites hold d_1.

3. We recursively copy the contents of d_n into the writer site. In the case of dedupFS, into directory "2" of the local dedupFS mount directory.

4. We again synchronize the reader site with the writer site. In this moment, both sites hold d_1 and d_n.

Concurrent workloads. Concurrent workloads illustrate multiple-writer scenarios. In this case, two writer sites, *a* and *b*, start with a common, initial version of the workload directory, d_1. Each writer site then independently modifies its local replica of the workload directory, respectively producing versions d_a and d_b. After such a concurrent update activity, both sites synchronize their replicas.

More precisely, the procedure is the one that Figure 8.21 presents:

[6] The option of a two-version analysis was also taken for the evaluation of most relevant related work, e.g. [MCM01, JDT05, ELW+07, Hen03].

[7] Again, this is the same experimental procedure that relevant related work, such as [MCM01, JDT05, ELW+07, Hen03], follows.

Workload	Type	#Files	d_1 size (B)	d_n size (B)
`course-docs`	Concurrent	9,485	29,567,088	82,480,704 (d_a); 43,418,272 (d_b)
`student-projects`	Concurrent	6,487	22,643,346	30,160,055 (d_a); 9,083,818 (d_b)
`gcc`	Sequential	42,036	142,105,997	173,078,627
`emacs`	Sequential	4,210	45,988,215	54,730,114
`linux`	Sequential	28,573	156,568,220	162,381,404
`usrbin`	Sequential	8,101	201,681,977	333,910,547

Table 8.7: Overall characteristics of evaluated workloads.

1. We recursively copy the contents of d_1 into directory "1" of *A*'s mount directory.

2. We synchronize *B* with *A*. In this moment, both sites hold d_1.

3. We recursively copy the contents of d_a into directory "2a" of *A*'s mount directory. In parallel, we recursively copy the contents of d_b into directory "2b" of *B*'s mount directory.

4. We synchronize *B* with *A*. In this moment, *B* holds d_1, d_a and d_b.

The contents of d_a and d_b can reflect out-of-bound interactions that have passed unnoticed by dedupFS while each site concurrently produced each such divergent version, as Figure 8.21 illustrates. Notice, however, that such interactions are already implicit in the contents of d_a and d_b and, therefore, do not constitute any explicit step of the experiment.

8.3.1.2 Workloads

Workload selection was initially driven by the standard workloads that related data deduplication literature uses to evaluate proposed solutions. In particular, we have closely followed the set of workloads adopted by Jain, Dahlin and Tewari when evaluating the TAPER protocol [JDT05]. Their workloads reflect a wide range of realistic usage scenarios. Furthermore, with rare exceptions[8], their set of workloads is representative of the workloads found in other related work.

The standard workloads consist of sequential workloads. In order to attain a more complete evaluation, we introduce two concurrent workloads, obtained from real data.

Table 8.7 presents the considered workloads along with their overall characteristics, including number of versions, number of files and plain sizes. Every workload comprises considerably large overall contents (ranging from more than 100 MB up to more than 2 GB) and file sets (ranging from more than 850 files up to more than 42,000 files). The large dimensions of the considered workloads

[8] For instance, [ELW+07] uses animation rendering data snapshots. We could not use such a workload, since it was not publicly available.

prove the robustness of the current dedupFS prototype implementation, which was able to correctly store and synchronize such large file sets. At the same time, such large samples lend strong statistical value to the results that we present next.

We may divide the workloads into the following three categories:

1. Collaborative document editing.

   This category is the closest to the collaborative scenarios that the thesis mainly targets. Furthermore, it is the only category to include concurrent workloads.

   Two workloads exist in this category. We obtained both from real data from the Distributed Systems (DS) and Software Engineering (SE) undergraduate courses of the Instituto Superior Técnico faculty of Lisbon. Data spans across a complete 6-month semester.

   A first workload, called `course-docs`, consists of snapshots of the CVS repositories used by the lecturers of both courses to keep pedagogical material (e.g. tutorials, lecture slides, code examples and project specifications) and private documents (e.g. individual student evaluation notes and management documents). The type of files varies significantly, ranging from text files such as Java code files or html files, to binary files such as pdf documents, Java libraries, Microsoft Word documents and Microsoft Excel worksheets.

   Both courses were tightly coordinated, since their students are partially evaluated based on a large code project that is common to both courses.[9] Consequently, lecturers of both courses carried an eminently collaboratively activity, involving regular meetings.

   The lecturers maintained two CVS repositories: a common DS&SE repository, which lecturers of both courses accessed to share documents related to both courses; and a DS-only repository, which only DS lecturers used.

   The `course-docs` workload starts (d_1) with a snapshot of both repositories at the beginning of the semester. In this moment, both repositories have old documents from previous years. Such an initial version then diverges to two concurrent versions. A first one, d_a, contains a snapshot of the DS-only repository at the end of the semester; a second one, d_b includes the new contents of the common DS&SE repository at the end of the semester.

   We believe that this workload is an very good case to evaluate the amount of out-of-band redundancy. The concurrent evolutions from d_1 to d_a and d_b depend on data that was copied from d_1, which dedupFS is able to detect and exploit. However, the users of both repositories had regular out-of-band interactions, such as frequent email exchanges, and weekly meetings. Such

[9] The difference being that each course evaluates different aspects of the project that students deliver.

out-of-band interactions allow new data (i.e. not originally present in d_1) to be exchanged between d_a and d_b outside the control of dedupFS. By studying this workload, we are able to quantify the amount of such out-of-band redundancy.

A second workload, `student-projects`, captures the collaborative work among students of both courses. Teams of 9 students were given the assignment of developing a code project in Java, which they were demanded to develop on a CVS repository. To ease their coding effort, they were provided with a bundle of auxiliary Java libraries and illustrative code examples. Most teams relied on this bundle as a basis from which they started developing their final project.

The project lasted for 3 months. The workload starts (d_1) with a snapshot of the initial library/code bundle that was made available to every student. We then randomly selected two final projects from the CVS repositories, respectively calling them d_a and d_b. The snapshots consist mainly of binary Java libraries, Java code files and text-based configuration files, as well as Microsoft Word and pdf documents.

Similarly to the `course-docs` workload, the collaborative scenario captured by `student-projects` had sources of out-of-band redundancy. During the 3 months, the students were provided with additional code examples via the courses' web sites. The students could freely download and use such examples in their projects; such a web source is not traceable by dedupFS, hence contributing to out-of-band redundancy. Furthermore, the two teams had weekly laboratory courses together, where they discussed their current solution and borrowed possibly similar ideas and code samples from a common teaching assistant.

We have also evaluated the previous concurrent workloads as sequential workloads. In this case, we considered the sequential transition from d_1 to d_a.

2. Software development sources.

   A second set of workloads focuses on collaborative code development scenarios. These workloads include, for different real-world open-source projects, two consecutive versions of their source code releases. Namely, we have selected the source trees of recent versions of the gcc compiler, the emacs editor, and the Linux kernel. The choice of projects and versions is the same as adopted[10] for the evaluation of the TAPER data deduplication scheme [JDT05]. Using similar workloads as those used with TAPER allows us to compare our results with theirs.

[10] With the exception of the rsync workload, which we omit because we were not able to find identical versions as those that [JDT05] consider.

The `gcc` workload comprises the source tree for GNU gcc versions 3.3.1 and 3.4.1. The `emacs` workload includes the source code for GNU Emacs versions 20.1 and 20.7. The `linux` workload contains the source tree of the Linux kernel versions 2.4.22 and 2.4.26. With the exception of some files in `gcc`, the source trees consist exclusively of ASCII text files.

3. Operating system executable binaries.

   The previous workloads contain a large amount of text-based files. This last workload, `usrbin`, considers binary files, which have very different characteristics when compared with text-based files; e.g. the achievable data compression rates and cross-file and cross-version data similarity tend to be substantially lower for binary than text-based files.

   The `usrbin` workload includes the full contents of the `/usr/bin` directory trees of typical installations of the Ubuntu 6.06 and 7.10 64-bit Linux distributions. The `/usr/bin` directory contains most of the executable code binaries that are bundled with a Linux installation. Many of the executable files in the `/usr/bin` directories of versions 6.06 and 7.10 have common names; however, most of them differ in the source code version from which they originate, or in the compilation options that produced the code binaries.

8.3.1.3 Evaluated Solutions

We evaluate dedupFS with and without zlib compression. We considered different expected chunk sizes, namely 128 B, 2 KB and 8 KB. The maximum and minimum chunk size parameters were set in function of the expected chunk size; the maximum size was twice the expected chunk size, while the minimum size was a quarter of the expected chunk size.

In dedupFS, we used a directory cache of 150 directories, where each cached directory includes a maximum of 10,000 entries. Further, the size of the index array of the in-memory chunk hash table is 1024 elements. Variations to directory cache size and hash table index array size affect the performance of dedupFS. However, in order to limit our analysis to an acceptable number of variables, we assume the previous parameters as constant in the remainder of the section. We believe that the chosen values are reasonable for most devices that dedupFS targets.

We compare dedupFS with a relevant set of state-of-the-art alternative solutions to data deduplication. Every approach to data deduplication that Section 2.7 mentions is represented by at least one solution. Moreover, we also consider the basic approaches of plain transference and compression-only transference.

Table 8.8 sums up the characteristics of each solution evaluated, which we detail next.

- *rsync*, a Linux tool that employs compare-by-hash with fixed-size chunks [TM98]. We used version 2.6.3 of rsync. For a fair comparison with dedupFS, we disabled data encryption. We evaluated transference both with and

Solution	Approach	Version	Options
rsync	Compare-by-hash	2.6.3	`-rz`
TAPER	Compare-by-hash	results in [JDT05]	
dedup-lbfs	Compare-by-hash	protocol described in [MCM01]	
diff	Delta-encoding	2.8.1	`-ruN`
xdelta	Delta-encoding	1.1.3	`-n -0`
version-control-only	Version control		
plain	None		
gzip	Data compression	1.3.5	

Table 8.8: Overall characteristics of evaluated workloads.

without zlib compression [Deu50] (`-r` and `-rz` options, respectively). When compression was on, we used the default compression level of zlib (level 6), the same zlib level used with dedupFS. We used the default fixed chunk size, 700 B. For data volume measures, we resorted to information output by the `--stats` option.

The sender site ran an rsync daemon, with access to the version directories of some workload. The experiment consisted of running rsync at the receiver site, each time requesting one workload directory version (d_1, d_n, etc) to a target directory in the sender file system. Therefore, successive synchronization sessions took advantage of similar contents that already existed in the target directory, belonging to the previously downloaded version directory.

- *TAPER*, a compare-by-hash-based distributed data deduplication protocol [JDT05]. Although TAPER is not publicly available, their authors have published experimental results [JDT05] taken with some of the sequential workloads that we consider. For some of such measures, we were able to replay the same experiments with the same exact workloads, therefore obtaining results that allow a meaningful comparison with TAPER.

  The results used with TAPER were obtained with an expected chunk size of 4 KB and a maximum chunk size of 64 KB for TAPER's variable-size chunk phase (Phase II [JDT05]). TAPER further complements such a phase with three other phases, as Section 2.7.4.3 describes.

- *dedup-lbfs*, our implementation of the compare-by-hash protocol with variable-sized chunks used by the Low-Bandwidth File Solution (LBFS) [MCM01], (as Section 7.2.3 describes). We do not directly evaluate LBFS because no public stand-alone implementation of the original solution was available.[11]

[11] In fact, the implementation of LBFS is partially available inside the SFS [Maz00] framework. However, such an implementation does not allow running LBFS as a stand-alone file solution, as described in the original paper. In particular, it uses a cryptographic transport, which adds a significant overhead to the synchronization performance of LBFS. For this reason, we chose to re-implement LBFS using dedupFS as a base framework.

The experimental methodology with dedup-lbfs was analogous to the one we followed with dedupFS.

- *diff*, the popular Linux tool relying on delta-encoding. diff detects similarities between two text files, outputting a sequence of *added*, *deleted* and *changed* text extracts. Such extracts encode a space-efficient delta that is sufficient to reconstruct the second file from the first file.

  diff does not support distributed data deduplication by itself. However, diff can be easily complemented with some simple synchronization protocol, such as the one from the initial solution we describe in Section 3.2.4, to offer distributed data deduplication. With measurements taken with stand-alone diff, we are able to evaluate the relevant aspects of one such hypothetical distributed solution.

  We used version 2.8.1 of diff. We recursively applied diff between the root directories of d_1 and d_n (`-ruN` options). We set diff for highest precision in similarity detection (`-d` option). Since diff only works with text files, we only consider diff with the workloads that exclusively include text files.

- *xdelta*, a Linux tool for archiving and compressing consecutive versions of a file [Maca]. Similarly to diff, xdelta uses the delta-encoding approach. It runs a different algorithm to detect similarities between two consecutive versions of a given file [Maca]. xdelta then encodes the second version of the file as a delta file, consisting of a sequence of *insert* and *copy* instructions.

  Similarly to diff, xdelta does not support distributed data deduplication if used stand-alone. We used the same procedure as the one used with diff to partially evaluate an hypothetical solution that leveraged a simple synchronization protocol with xdelta. One fundamental difference to diff is that xdelta is able to compute deltas for both text and binary files. Hence, we evaluate xdelta with all workloads that this section considers.

  We used version 1.1.3 of xdelta, both with and without zlib compression of deltas (again, we used the default zlib compression level). We disabled xdelta's option of MD5 checksums in delta files for a fair data volume comparison of with other solutions.

- *version-control-only*, which consists of a simple shell script that approximately emulates synchronization protocols such as CVS's [C+93] or the initial solution's (see Section 3.2.4). Essentially, the script recursively compares two workload directory versions (e.g. d_1 and d_n) file-by-file and filters out the files with similar relative pathnames under each workload directory version that have identical contents. If some version tracking took place, the synchronization protocol would only propagate the remaining files, and ignore the unmodified files (which would have the same version in both workload directory versions).

This script is useful since it tells us the data volume that we would be able to avoid propagating during synchronization by tracking version histories, without resorting to data deduplication or compression.

- *plain*, plain transference of files. This alternative consisted of dedupFS with the deduplication protocol disabled.

- *gzip*, direct transference of zlib-compressed files [Deu50]. The same as above, with the difference that each file is compressed before being transferred.

8.3.2 Sequential Workloads

In a first experiment, we ran each sequential workload with every solution (and corresponding variants). Since out-of-band redundancy may only arise from concurrent work (as Chapter 6 notes), sequential workloads have no out-of-band redundancy. Hence, the following results evaluate the effectiveness of the considered solutions when detecting in-band redundancy only.

Most of the sequential workloads that we evaluate next closely resemble the very workloads that recent literature on data deduplication uses as motivational scenarios and based on which their authors advocate their solutions. More precisely, `gcc`, `emacs`, `linux` and `usrbin` are either the same workloads or very similar equivalents of the ones used by [MCM01], [JDT05] and [Hen03], for example.

Before actually experimenting with different solutions, we deduce some intrinsic compressibility and redundancy characteristics of each sequential workload we consider. Namely:

1. A first value is given by the zlib-compressed size of d_n, relative to its plain size; this constitutes a good estimate of the data compressibility of each workload.

2. A second value was obtained by dividing the file contents of d_n into chunks, using LBFS's algorithm with an expected chunk size of 128 B, and adding such chunks to a repository that already held the chunks of d_1. Then, we measured the size of the chunks of d_n that didn't yet existed in the repository, i.e. which were not redundant across d_1 and d_n. This value, obtained with an efficient algorithm using a relatively low expected chunk size, is a strong estimate of the data redundancy of d_n across d_1 and d_n.

3. Finally, a third value indicates how much workload contents reside in files whose contents and names remain the same from d_1 to d_n. We calculated such a value by running `diff` between files with similar relative pathnames from both directories, d_1 and d_n. This value helps us understand the write patterns that produce d_n. From it, we can distinguish whether write activity is concentrated in a small number of files of spread across many files.

Workload	Compressed Size	literal Chunks	Size of Untouched Files
course-docs	69,7%	55,3%	5,9%
student-projects	31,5%	57,1%	0,0%
gcc	24,0%	44,3%	11,6%
emacs	28,7%	46,4%	12,4%
linux	25,5%	11,1%	61,5%
usrbin	42,8%	71,6%	0,2%

Table 8.9: Analysis of evaluated sequential workloads, regarding compressibility, touched files and chunk redundancy. Compressibility assumes zlib's data compression algorithm, while chunk redundancy assume LBFS's algorithm with an expected chunk size of 128 B, applied locally.

Table 8.9 presents the previous values for each sequential workload. It shows that the set of sequential workloads we study covers a wide space of workloads, whichever axis we consider. The majority of workloads are highly compressible, in contract to `course-docs` and `usrbin`, which exhibit relatively low compressibility. Workloads such as `student-projects` and `usrbin` have no or almost no file that remains unmodified, while in `linux` only less than 40% of file content belongs to files that were actually written. Finally, most workloads have relatively high redundancy across their contents (42% to 55% in redundant chunks). `usrbin` is the negative exception, with less than 30% of redundant content, while `linux` sets the redundancy record with 88%.

Before proceeding with the evaluation of each solution, it is pertinent to discuss the effective meaning of an analysis of sequential workloads. In theory, the conclusions that we take from the results we present next are inevitable incomplete. For, in an optimistically replicated system, concurrency across replicas can always occur and, consequently, out-of-band redundancy may arise.

However, provided that, in practice, such out-of-band redundancy is negligible when compared to in-band redundancy, the present analysis is completely valuable to evaluate dedupFS. For the moment, we assume such a premiss: that out-of-band redundancy only arises at negligible levels (when compared to the in-band redundancy). Section 8.3.4.1 will then try to validate such a premiss with real concurrent workloads.

8.3.2.1 Transferred Volumes

We start by analyzing how much data and meta-data each solution needs to transfer to bring a reader site, initially holding d_1, up-to-date with a writer site holding both d_1 and d_n. Figures 8.22 to 8.24 present such measures for each sequential workload and solution.

For most solutions, we are able to distinguish between file content data and meta-data such as directory entries, version vectors or chunk references. In the case of diff and xdelta, we were not able to make such a distinction, and hence we

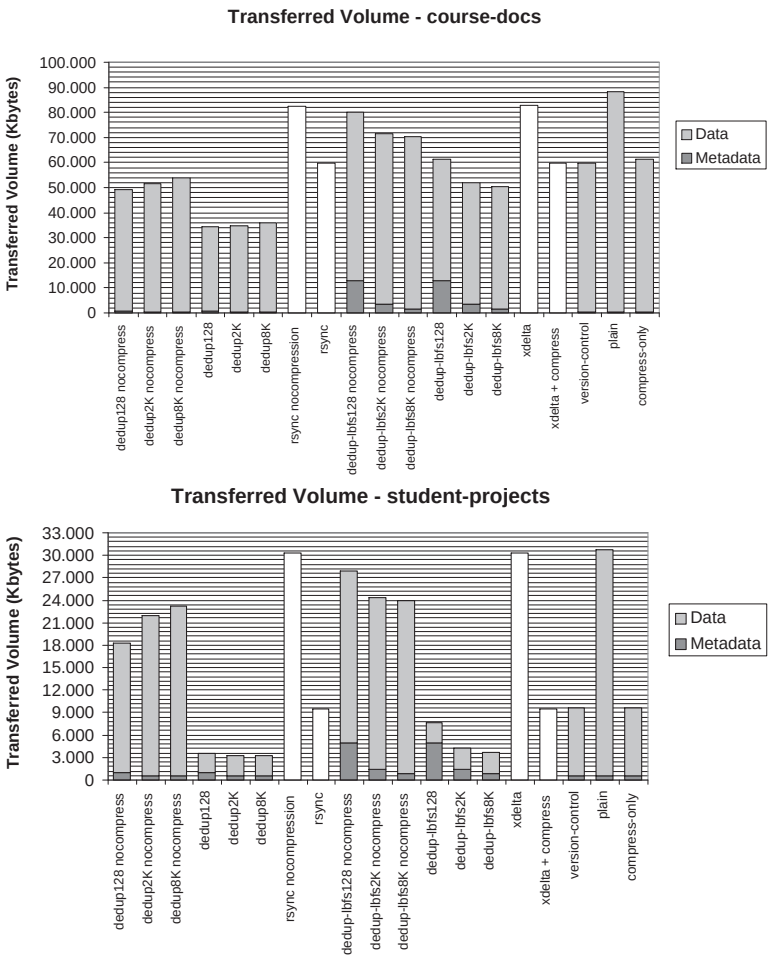


Figure 8.22: Transferred data volumes for collaborative document editing workloads.

do not differentiate their transferred volume. Furthermore, in the case of TAPER, the meta-data only considers information exchanged by the deduplication protocol, lacking the overhead of directory meta-data of TAPER.

In what follows, we follow a bottom-up analysis of dedupFS. We start by focusing on dedupFS only, comparing the different its evaluated instances. We then turn our attention to compare-by-hash and delta-encoding solutions, comparing them with dedupFS. Finally, we consider plain and gzip-compressed transference.

dedupFS with varying expected chunk sizes, with and without data compression. We start by focusing our attention on the different dedupFS instances, before comparing dedupFS with other solutions. The evaluated dedupFS instances include dedupFS with different expected chunk sizes, namely 128 B, 2 KB and 8 KB, both with zlib data compression turned on and off.

As expected, the results confirm that smaller chunk sizes yield higher precision in detecting redundancy and therefore achieve lower transferred data volumes. In fact, such a decrease is substantial for all workloads except for the binary workload, `usrbin`. This suggests that redundancy across binary executables mostly occurs at coarse-grained chunks; as decreasing expected chunk size from 8K to 2K, and then to 128 B, detected only 2,4% and 2,9% more redundant data, respectively.

On average (over all workloads), dropping from 8 KB to 2 KB expected chunk size results in more 17,1% data detected as redundant, while further decreasing to 128 B expected chunk size finds more 15,0% redundant data (relative to 2 KB expected chunk size). Turning data compression on attenuates such differences to 15,1% and 11,0%, respectively. Such an effect is natural as, with finer-grained chunks, less literal data is compressed, hence less gains are achieved from data compression.

Evidently, literal data volume is not sufficiently meaningful to evaluate dedupFS, as it neglects the overhead with chunk references that dedupFS introduces. If we take such an overhead into account, the gains in transferred volume, relative to the gains in data volume, inevitably drop.

Figure 8.25 takes a closer look at such gains for every workload, confronting them with the number of chunk references that dedupFS transmits. From it, we can see that reducing expected chunk size from 8K to 2 KB, and then to 128 B, yields more 15,4% and 12,4% reductions in transferred volume, respectively (in contrast to 17,1% and 15,0% reductions in data volume, respectively).

Most importantly, reducing chunk size is always beneficial in terms of transferred volume. The reference coalescing step of dedupFS (see Section 6.4.2) is key in ensuring such an important condition. An interesting observation is that, when data compression comes into play, reference coalescing may no longer ensure the previous condition. The `student-projects` workload exposes such an evidence: when going from 8K- to 2 KB expected chunk size, with data compression active, the latter actually transfers more 9% volume than the larger chunk alternative.

The reason behind this pathological effect is the difference in compressibility between chunk references and the chunk contents that the former replace. Since reference coalescing does not take data compressibility into account, it ignores the fact that dedupFS does not zlib-compress chunk references, while it does compress chunk contents.[12] Hence, opting for a smaller volume of chunk references instead of a larger volume of chunk contents may not always be the best choice when we enable data compression.

Although not as evident as in the `student-projects` workload, the previous effect is present in all the other workloads. With data compression on, when we drop from 8 KB expected chunk size to 2 KB, and from 2 KB to 128 B, the gains in transferred volume become 10,2% (in contrast to 15,4% without data compression) and 3,0% (in contrast to 12,4% without data compression), respectively.

[12] Even if dedupFS did compress chunk references, the attained compression gains would be small, as chunk references are poorly compressible.

Comparison with compare-by-hash solutions. We now proceed to compare dedupFS with the remaining solutions, starting at those that adopt the compare-by-hash approach. Let us recall Figures 8.22 to 8.24. Unless where noted, we use dedupFS with 128 B expected chunk size, which we abbreviate as dedupFS-128, as the reference in what follows.

With few exceptions, which we address next, dedupFS attains significantly higher reductions in transferred volume than any compare-by-hash solution, for any workload. Essentially, the key reason for such an advantage is the fact that dedupFS matches similar chunks by running a highly accurate *local* algorithm on local file contents. Consequently, dedupFS is able to be highly accurate in redundancy detection while keeping meta-data volume exchanged during synchronization to low levels. This contrasts with the considered compare-by-hash solutions, which employ a *distributed* redundancy detection algorithm. Hence, they cannot avoid the trade-off between higher accuracy and more overhead with transferred deduplication they transfer across the network meta-data (namely, hash values and chunk references).

Needless to say, the present evaluation leaves out the unmeasurable advantage of dedupFS of not being prone to data corruption due to hash collisions. By definition, all compare-by-hash solutions we consider are vulnerable to such a possibility.

We take rsync as a first example. On average over all workloads, rsync transfers 35% more (data and meta-data) volume than dedupFS-128 when we disable data compression. Turning data compression on worsens rsync efficiency, as it transfers, on average, 48% more volume than dedupFS-128. Such a value suggests that rsync's current implementation applies zlib in a less effective manner (e.g. compressing individual chunk, instead of chunk streams).

Three main factors contribute to the difference in volume efficiency between rysnc and dedupFS-128. First, the increased deduplication overhead of rsync's distributed redundancy detection algorithm.[13] Second, rsync's algorithm works with larger chunks than dedupFS-128; otherwise, previously mentioned overhead would grow to unacceptable levels. Third, rsync's algorithm works exclusively on between pairs of files, matched by pathname. Hence, rsync is only able to detect redundancy across files with a similar pathname.

The third factor is probably the crucial factor for the differences in rsync'r efficiency with each workload. On the worst extreme, rsync drops to 66% and 67% more volume than dedupFS-128 with the `course-docs` and `student-projects` workloads (without data compression). This is easily explained by the frequent file renaming activity in these workloads.

On the other hand, in workloads where pathnames are practically stable (`gcc`, `emacs`, `linux`), rsync is able to reach closer to the dedupFS-128's levels of transferred volume. In particular, we found one exception, `linux` with compression enabled, where rsync actually transfers -17% volume than dedupFS-128 with com-

[13] Unfortunately, rsync does not output the size of such an overhead and, therefore, we are not able to quantify it.

pression enabled. The previous results of Jain et al. [JDT05] seem to explain such an exception. Jain et al. show evidence that, for cases of small changes randomly distributed in a file that does not change its pathname, rsync's similarity detection algorithm is more efficient than a variable-size chunk algorithm (on which dedupFS is based).

Note, however, that dedupFS may also employ rsync's algorithm for local similarity detection, or even combine it with dedupFS's current algorithm. More generally, dedupFS may adopt any distributed redundancy detection algorithm, employing it locally across the local contents at each site.

Another interesting analysis is to compare dedupFS with the higly optimized, 4-tiered distributed deduplication scheme of TAPER. Such a comparison is possible with the `gcc`, `emacs` and `linux` workloads, for which TAPER results are publicly available [JDT05]. Both solutions transfer comparable data volumes: on average over the three workloads, TAPER is only 0,29% better than dedupFS-128. However, distinguishing transferred volume between content and deduplication meta-data volume allows more meaningful conclusions.

On one hand, both solutions exhibit very similar accuracy in redundancy detection with some workloads. On average over the three workloads, TAPER detects more 0,14% more redundant data than dedupFS-128 (with a maximum of more 4,0% redundant data detected with `gcc`, and a minimum of less 2,9% with `emacs`).

On the other hand, TAPER's more intricate distributed algorithm generally requires transmitting more bytes of deduplication meta-data (more 21,3% than dedupFS-128, on average over the three workloads). Figure 8.26 illustrates such an advantage of dedupFS. It presents the overheads with deduplication meta-data for dedupFSand TAPER, as well for other solutions. For `gcc` and `emacs`, TAPER imposes substantially more deduplication overhead than dedupFS-128 (21,2% and 73,9%, respectively), in contrast to `linux`, where TAPER's scheme has 31,0% less overhead.

More importantly than the differences in deduplication overhead, TAPER 4-tiered algorithm exchanges such meta-data by a 4-round-trip protocol. If compared with dedupFS's single-round-trip protocol, the increased latency of the additional network round-trips may be a decisive disadvantage of TAPER. However, the present evaluation could not quantify such a disadvantage, as TAPER was not publicly available for testing.

Finally, we compare dedupFS with our implementation of LBFS'scheme, dedup-lbfs. dedupFS and dedup-lbfs employ practically similar algorithms for similarity detection, with some important differences. The results expose three such differences.

Firstly, dedup-lbfs, following LBFS's original algorithm [MCM01], does not exploit recursive redundancy (see Section 6.2.2), i.e. redundancy that may exist exclusively across the contents to send. dedupFS incorporates such an optimization. Its effects are significant: they are evident on the differences between the data volume that each solution considers as redundant. On average over all workloads, such an optimization detects 10,8%, 14,4% and 20,8% more redundant chunk con-

tent for expected chunk sizes of 8K, 2K and 128 B.

We should note that the recursive redundancy optimization is not exclusively applicable to dedupFS. One can directly incorporate it LBFS's original algorithm, hence eliminating the above differences in detected chunk contents.

The second relevant difference is that dedupFS applies the similarity detection algorithm locally across the local contents at each site. In contrast, dedup-lbfs applies the algorithm in a distributed manner across the new versions at the sender site and the contents that the receiver site holds. Consequently, dedup-lbfs needs to exchange the hash values of the chunks that comprise the versions to propagate. This is not the case with dedupFS.

It is easy to show that, theoretically, the number of hash values that dedup-lbfs transmits grows linearly as the expected chunk size decreases. Hence, as one decreases the expected chunk size from 8 KB to 2 KB or 128 B, the extra accuracy comes at the expensive cost of a linear growth of overhead with hash values. This is a well known result [MCM01, JDT05, ELW+07, BJD06].

In practice, this cost is worsened by the fact that the small dimensions of most files in typical file system workloads [Vog99] (including the ones we consider). Our evidence shows that small files bias the actual chunk sizes to at most 2 KB, no matter how high we set the expected chunk size. For example, the highest average chunk size that we found while dividing chunks using an expected chunk size of 8 KB was 2.5 KB (`student-projects` workload).

As a consequence, dedup-lbfs introduces a substantially larger overhead with deduplication meta-data that one might predict theoretically. Recalling Figure 8.26, such an increase is evident and grows as expected chunk size decreases. On average over all workloads (including `course-docs` and `student-projects`, not present in Figure 8.26), dedup-lbfs-8K exchanges 147% more meta-data than dedupFS-8K, fundamentally due to hash values. If we step down to dedup-lbfs-2K, such a difference rises to 387% more deduplication overhead than dedupFS-2K. Finally, dedup-lbfs-128 increases deduplication overhead by over an order of magnitude, relatively to dedupFS-128.

The impact of the higher overhead on the overall transferred volume (i.e. literal data plus deduplication meta-data) is relevant. This is true even if we assume that dedup-lbfs incorporates the optimization of exploiting recursive redundancy (by replacing the data volume measures of dedup-lbfs by dedupFS's). Under such an assumption, dedup-lbfs-128, dedup-lbfs-2K and dedup-lbfs-8K would transfer 35%, 7% and 2% more bytes than dedupFS-128, dedupFS-2K and dedupFS-8K, respectively, on average over all workloads.

Thirdly, since dedup-lbfs has no mechanism as dedupFS's reference coalescing step, it may easily transfer a larger volume than the plain approach. This is the case of the `usrbin` workload, where the low redundancy does not compensate for the overhead of exchanging hash values (see Figure 8.24). Note that, despite unobserved with our workloads, such a situation may also arise with the remaining compare-by-hash solutions we have evaluated. As noted before, such a situation cannot arise with dedupFS (except in rare cases if data compression is enabled, as

explained earlier in this Section).

Comparison with delta-encoding solutions. Interestingly, the delta-encoding solutions, diff and xdelta, proved to be strong contenders of dedupFS, with regard of transferred volumes. In half the workloads, delta-encoding was significantly more efficient in exploiting redundancy than any other solution, including dedupFS.

Namely, these are the workloads with high redundancy between the same files of d_1 and d_n (whose pathnames remain the same in both versions). More precisely, the most efficient delta-encoding solution, xdelta, would transfer 29%, 39% and 44% less volume than dedupFS-128 in `gcc`, `emacs` and `linux`, respectively.[14]

Delta-encoding solutions are specifically tailored to detect similarities between pairs of versions of the same file, which explains such an advantage. They employ special purpose similarity detection algorithms that are limited to running locally, and between two versions only. Furthermore, the delta encoding format [HHH+98] is lighter than the chunk references that dedupFS (and compare-by-hash) exchanges. For instance, the former takes advantage of the fact that the referenced version identifier is already implicit; hence, it needs not be explicitly encoded.

When file renaming and copy become frequent, or redundancy across multiple versions/files becomes significant, delta encoding looses its efficiency to dedupFS. Namely, for the remaining three workloads, `course-docs`, `student-projects` and `usrbin`, xdelta transfers 68%, 66% and 33% more volume than dedupFS-128, respectively. Moreover, in such workloads, xdelta performed worse than any compare-by-hash solution.

We may regard delta encoding as a particular instance of the versioning-based deduplication approach that dedupFS follows. The difference being that delta encoding is a more constrained solution, which may only exploit redundancy between pairs of consecutive versions of the same object (i.e., files in this context). If most redundancy occurs with such a pattern, delta encoding can be very efficient, as some of our results suggest. However, dedupFS's ability to exploit multiple cross-version and cross-object redundancy relations allows dedupFS to adapt well to general patterns of file system usage.

As a final remark, the previous conclusions suggest that a possible direction for future work would be to leverage dedupFS with xdelta's algorithm and encoding format in situations where redundancy occurs between consecutive versions of the same file.

[14] Such an advantage drops to 15%, 27% and 44% when we enable data compression in both solutions. However, the most recent version 3 of xdelta claims to have a more efficient, self-contained data compression algorithm, specifically designed for the VCDIFF encoding format [Macb].

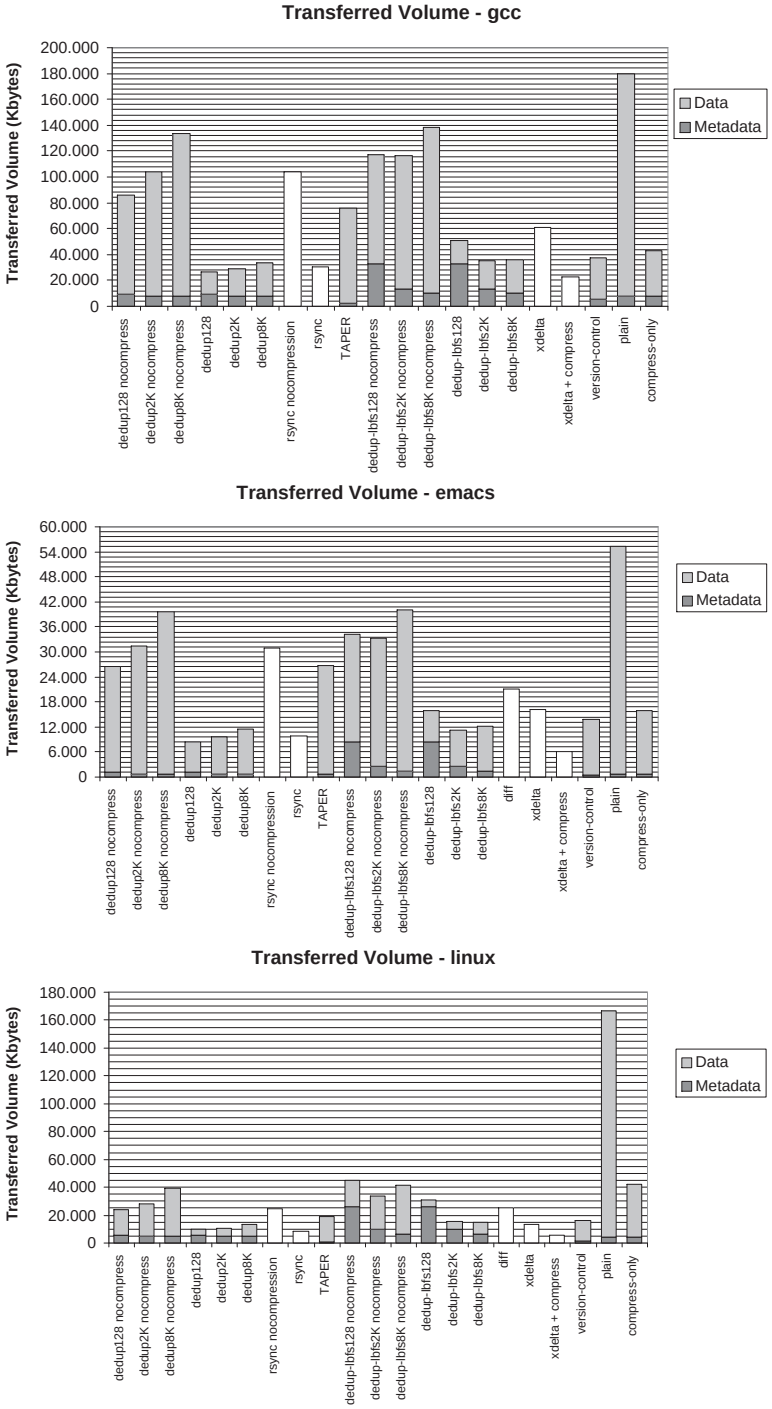


Figure 8.23: Transferred data volumes for software development workloads.

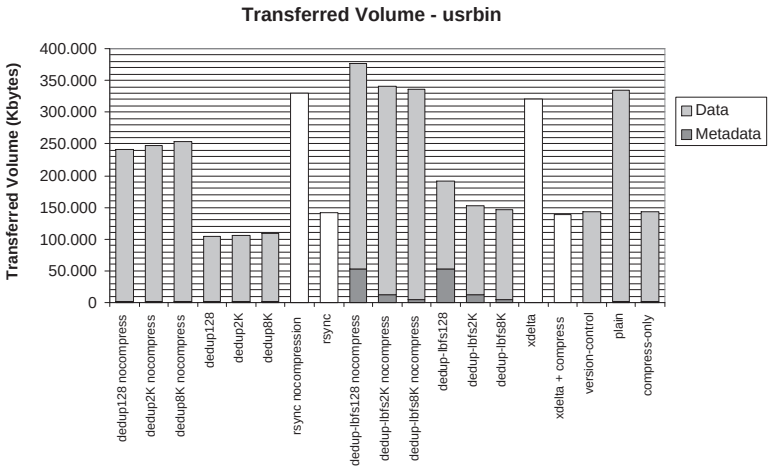


Figure 8.24: Transferred data volumes for the executable binaries workload.

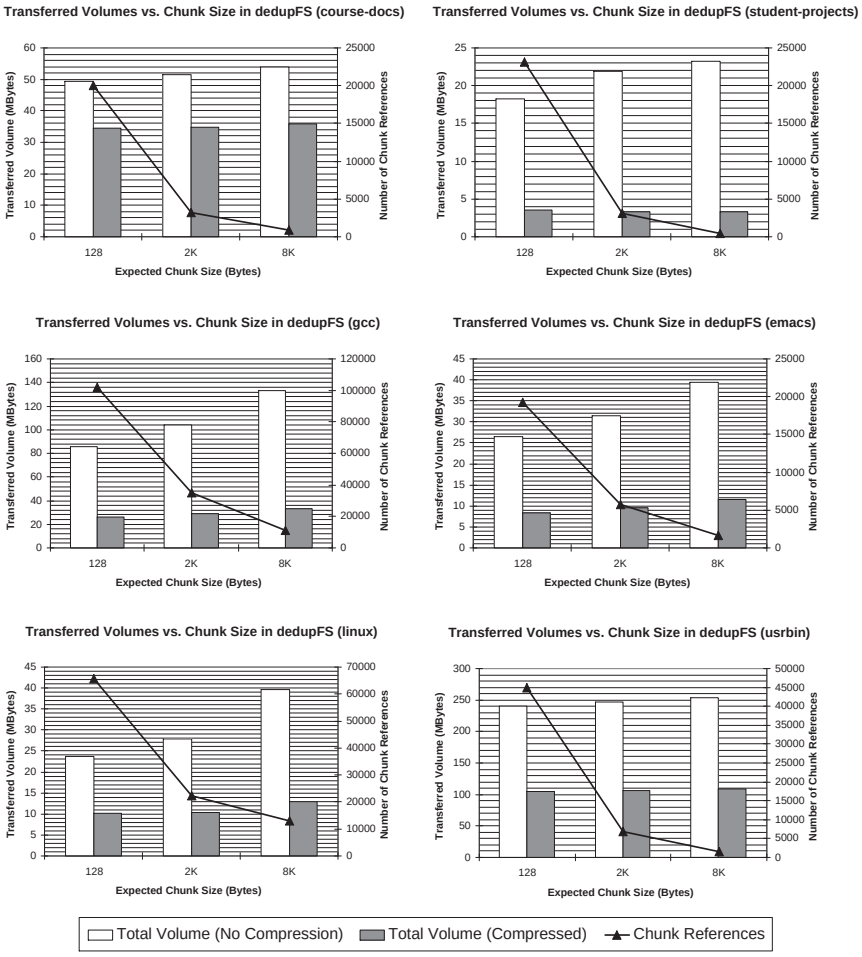
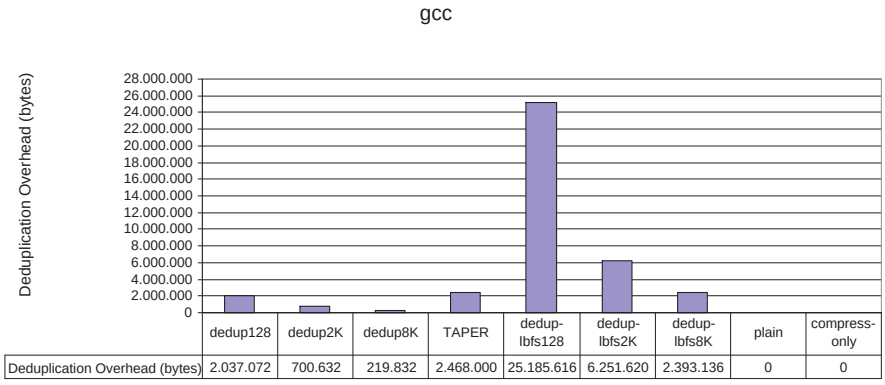
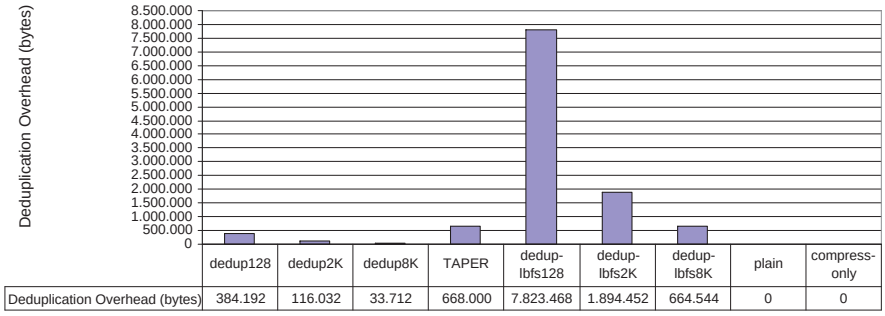


Figure 8.25: Transferred volumes vs. chunk size in dedupFS





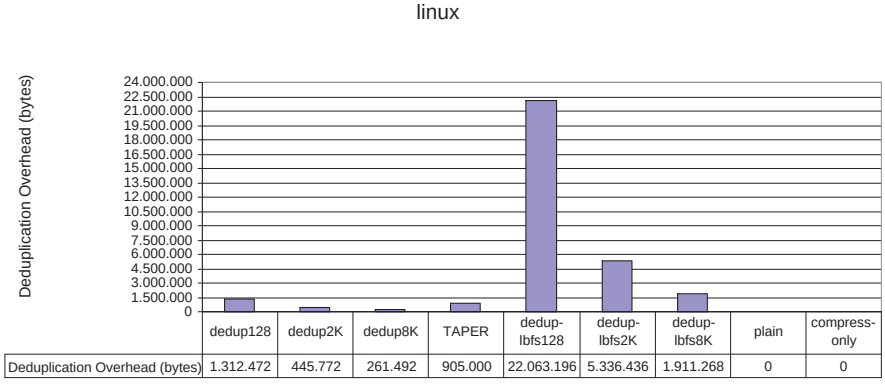


Figure 8.26: Transferred volume overheads due to deduplication (i.e. transferred meta-data such as hash values and chunk references).

8.3.3 Synchronization Performance

Measuring and comparing transferred volumes gives us a valuable estimate of the performance of each solution. However, low data and meta-data volumes does not necessarily mean good performance. Factors such as number of network round-trips and local processing time may also strongly affect its overall performance.

As a second experiment, we set out to measure the time that each solution takes to complete synchronization. This experiment does not include some solutions, namely: TAPER, for which a running executable was not publicly available; diff and xdelta, which do not support distributed synchronization in their current implementation.

The experiment used a 100Mbps local ethernet connection between the sender and receiver sites. Such a bandwidth is one or two orders of magnitude higher than the typical bandwidth in the environments that dedupFS mainly targets, namely the Internet and mobile networks. From the viewpoint of dedupFS, we may regard such a setting as a *near-worst-case* scenario. It conceals the positive impact of the reductions in transferred volume (which we have already evaluated in the previous section), while amplifying the performance overhead of the additional processing.

Figures 8.27 to 8.29 present the performance results for every sequential workload. For all solutions except rsync, we were able to differentiate two components of the total synchronization time: a pre-download time, which includes all protocol steps before the actual download of the contents of the new versions starts; and the remaining, download time.

On average over all workloads, dedupFS-2K with no compression exhibits the best performance among all dedupFS variants. However, turning data compression on does compensate on some highly compressible workloads: dedupFS-8K+zlib takes 40,2% less time to synchronize than dedupFS-2K with no compression in the `student-projects` workload, while dedupFS-8K+zlib and dedupFS-128+zlib achieve marginal performance gains in the `gcc` and `linux` workloads, respectively.

In spite of the high bandwidth, the download times of dedupFS dominate pre-download times (download time takes more than 80% of the total synchronization time) for the majority of workloads.

Some pathological cases of excessive pre-download time occur, which expose a weakness of dedupFS's current design and implementation, which we discuss as follows. dedupFS's optimization of exploiting recursive redundancy is, in its current implementation, an inefficient step. Currently, such a step implies, for every chunk reference of every version to send, traversing a singly-linked list that holds the identifiers of the versions to send that the pre-download phase has already handled.

Hence, the performance penalty of such an optimization grows quadratically with the number of versions/objects to send at a single synchronization session. Notwithstanding the significant reductions in transferred data volume of such an optimization (see Section 8.3.2.1), such a penalty has a strong impact in the workloads where d_n comprises a large number of files. This is the case of the `gcc` and

`linux` workloads (for any variant of dedupFS) and the `student-projects` workload with dedupFS-128, where dedupFS exhibits excessive pre-download times that range from 38% to 89% of the total synchronization time.

Disabling the detection of recursive redundancy considerably reduces the pre-download times in the above pathological cases. For instance, the pre-download time of dedupFS-128 (with no compression) drops from 7.476 ms to 441 ms. As an inevitable consequence, transferred data volume rises from 17.2 MB to 22.9 MB. Such an increase, however, has no significant impact on the overall performance with 100Mbps. It may, of course, become relevant if one considers environments of lower bandwidths.

Eliminating the performance penalty of recursive redundancy detection is, therefore, a key goal of future work. A potential solution must consider more efficient membership inclusion tests, rather than the current list traversal test.

When compared to the other solutions, dedupFS achieves better performance, with a small number of exceptions in some of the pathological workloads that we mention above. The advantage over rsync is substantial. On average over all workloads, rsync almost doubles dedupFS-2K's total time (91.7% and 105.8% slower with zlib compression off and on, respectively). Even in the worst workloads for dedupFS, rsync is at least 40.8% slower than dedupFS-2K. It is worth noting that rsync never accomplishes better performance than the plain transfer alternative.

Concerning dedup-lbfs, its performance degrades considerably as we decrease the expected chunk size. With an expected chunk size of 128 B, dedup-lbfs reflects the strong penalty of a large network overhead with hash values and chunk references and, most importantly, the delay of numerous lookups to the chunk hash table. Its synchronization time is, on average over all workloads, 17 times slower than plain transfer and 26 times slower than dedupFS-2K.

Its performance rises substantially as we increase expected chunk size to 8 KB. In that configuration, dedup-lbfs is, on average over all workloads, 29.5% slower than dedupFS-2K. More notably, dedup-lbfs-8K outperforms dedupFS in the pathological workloads that we mention above: dedup-lbfs-8K is 11.6% and 11.7% faster than dedupFS-2K in `gcc` and `linux`, respectively. Nevertheless, if we disable recursive redundancy detection, dedupFS-2K becomes faster than any deduplication solution for every workload.

As a final remark, we compare dedupFS with the plain and compress-only alternatives. In general, dedupFS substantially outperforms the previous solutions, in almost all workloads with relatively high redundancy (on average, 25.6% and 27.8% faster than plain and compress-only, respectively). The exception is the `gcc` workload, where the performance penalty of the pathological cases we describe previously is sufficient to make dedupFS a worst alternative than the plain and compress-only alternatives (16.8% and 17.8% slower, respectively).[15] Again, disabling recursive redundancy detection eliminates such performance losses.

[15] Another exception is dedupFS-128 and dedupFS-128+zlib in the `student-projects` workload. In this workload, however, all other variants of dedupFS outperform both plain and compress-only.

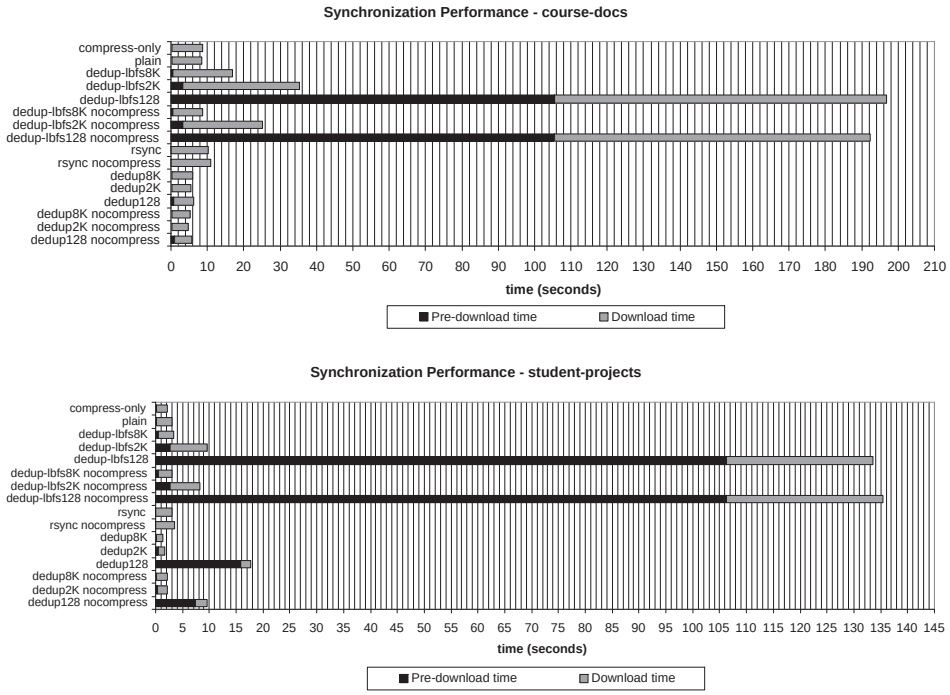


Figure 8.27: Synchronization times for collaborative document editing workloads.

When the redundancy degree is low, which is the case of the `usrbin` workload, dedupFS (with no data compression) is still able to attain better performance than the plain and compress-only solutions. However, the low compressibility of this workload (see Table 8.9) causes dedupFS+zlib to be slower than plain transfer, due to the additional processing time the former spends with zlib functions.

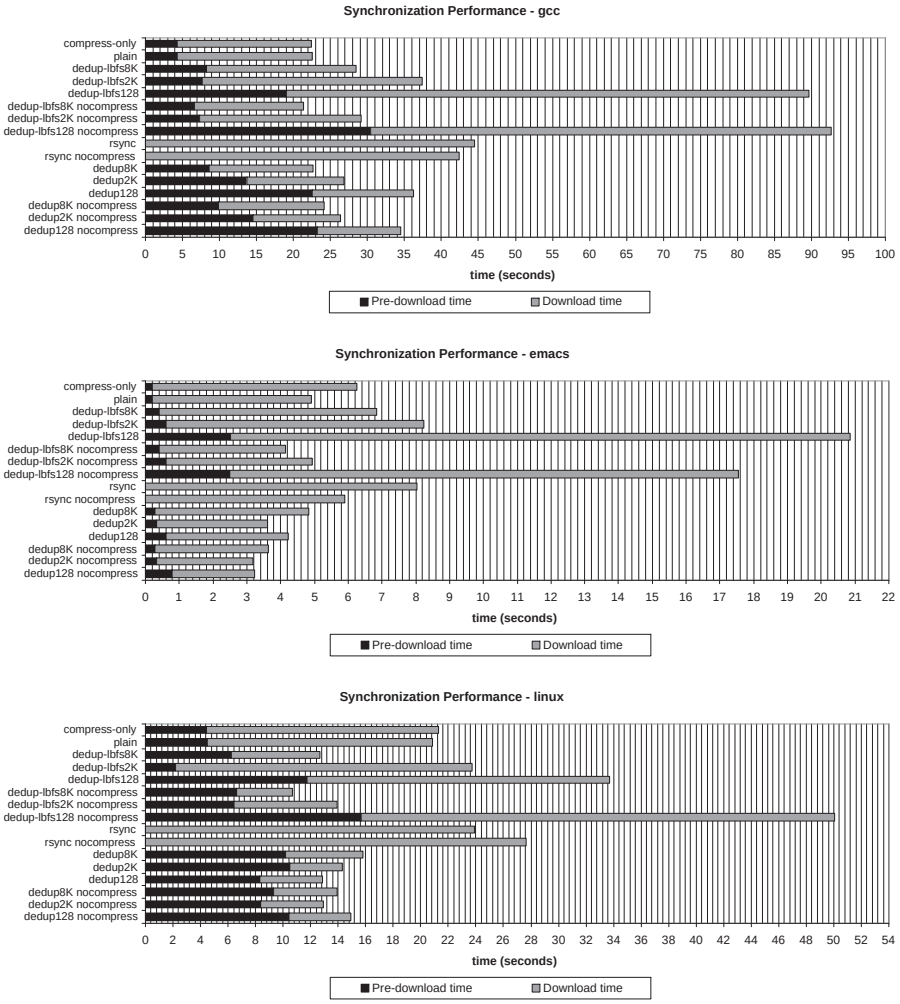


Figure 8.28: Synchronization times for software development workloads.

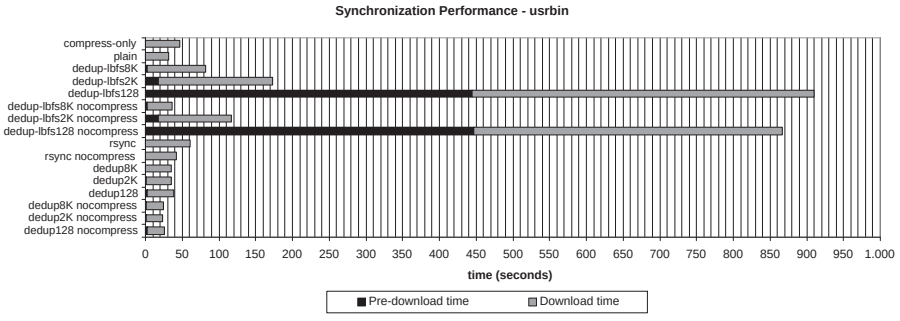


Figure 8.29: Synchronization times for the executable binaries workload.

8.3.4 Asserting Our Results with More Realistic Assumptions

As we explain in Chapter 6, the advantages of dedupFS over other solutions will only be effective as long as two conditions hold: (i) out-of-band redundancy, which dedupFS cannot exploit, is not significant; and (ii) the synchronizing sites are able to maintain sufficiently old version logs.

As we discuss before, the promising results in Section 8.3.2 do not completely evaluate dedupFS, as they exclusively consider two-version sequential workloads where out-of-band redundancy does not arise, and both sites' logs are assumed sufficiently deep to hold a common ancestor version (d_1). Hence, we can only accept the results from Section 8.3.2 as meaningful in situations where the two conditions above hold.

This section aims at understanding whether the scenarios that this thesis considers verify the two conditions. In the next two sub-sections, we gradually weaken the assumptions of the previous section. In Section 8.3.4.1, we evaluate concurrent workloads, where out-of-band redundancy finally occurs. Then, in Section 8.3.4.2 evaluates n-version workloads ($n > 2$) with version logs of limited size.

8.3.4.1 Out-of-band Redundancy

We measured the amount of out-of-band redundancy using real-world concurrent workloads, `course-docs` and `student-projects`. Both workloads result from collaborative activity among different teams of users, as we explain in Section 8.3.1.2. In both workloads, two sites started with a replica of a common, initial version. During a considerable period of time – 6 months for `course-docs` and 3 months for `student-projects`–, both replicas evolved independently with frequent concurrent updates (typically, more than one update per day).

The experiment assumes that, during such periods, the underlying file system can only trace the local evolution from the initial version to the final version of each replica. As Section 8.3.1.2 details, considerable information was potentially exchanged between both replicas via channels that are untraceable by the underlying file system (e.g. dedupFS). Inevitably, such out-of-band exchanges may lead to out-of-band redundancy, which dedupFS cannot detect.

We measured the amount of out-of-band redundancy that each workload collected at the end of its lifetime with a fairly simple experiment. Recalling Figure 8.21 from Section 8.3.1, each replica holds, after such a period, version d_a and d_b, respectively (besides d_1). For each workload, we synchronized both divergent replicas, the receiver replica holding versions d_1 and d_a, and the sender replica holding versions d_1 and d_b. Firstly, we synchronized using dedupFS-128[16], where the sender replica could only exploit in-band redundancy; i.e. the chunks that are redundant across d_1 and d_b. We then repeated the same experiment using dedup-

[16]We disabled recursive redundancy detection. Without such an optimization, dedupFS attains exactly the same efficiency in decting in-band redundancy as dedup-lbfs, for the same expected chunk sizes.

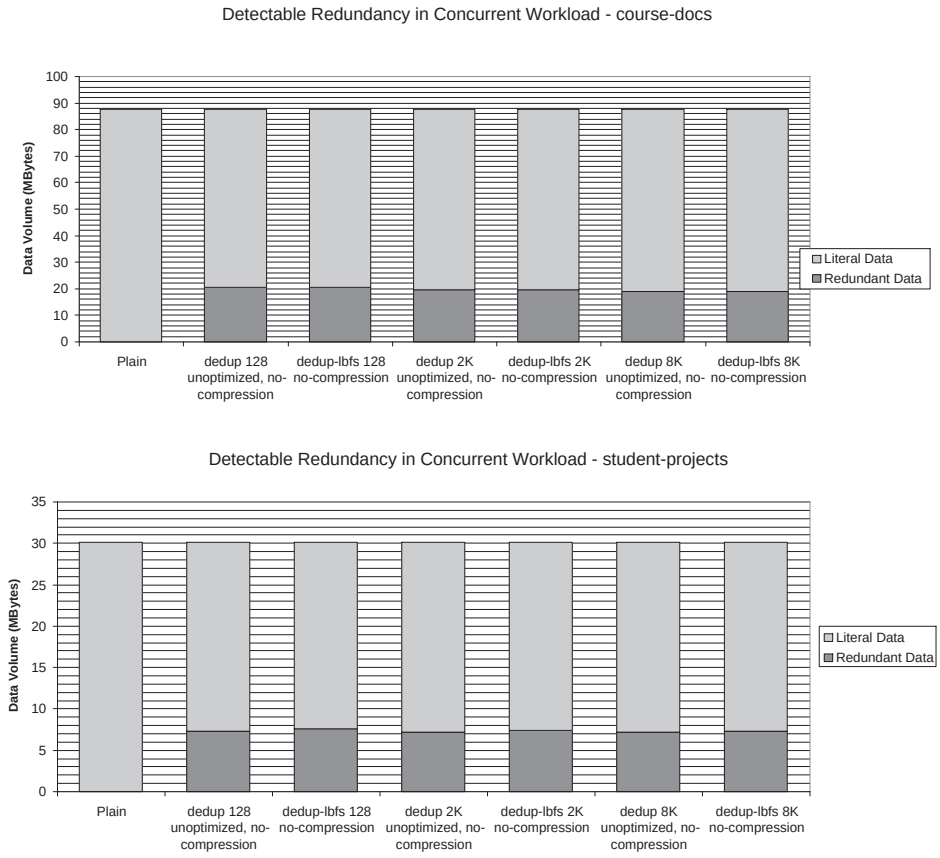


Figure 8.30: Differences in detected redundant data volume in the concurrent workloads (`course-docs` and `student-projects`), between a solution that can only detect in-band redundancy (dedupFS) and a solution that can detect both in-band and out-of-band redundancy (dedup-lbfs).

lbfs-128. In this case, the distributed redundancy detection protocol can detect both in-band and out-of-band redundancy; i.e. the sender can exploit chunk redundancy across any workload versions, d_1, d_a and d_b. Therefore, by subtracting the redundant data volumes[17] of dedupFS from dedup-lbfs, we obtain an estimate of out-of-band redundancy in the workload.[18]

Figure 8.30 shows the results for both concurrent workloads, considering different instances of dedupFS and dedup-lbfs, as well as the plain transfer solution. For simplicity, and without lack of generality, we disabled data compression in all solutions.

[17] I.e., without accounting for exchanged meta-data.

[18] Of course, the previous values are estimates because neither dedup-lbfs's nor dedupFS's redundancy detection algorithms are optimal. Nevertheless, using small chunk sizes (namely, 128 expected chunk size), we obtain an accurate estimate.

Perhaps surprisingly, the redundant data volume that results from out-of-band data exchanges during several months is almost insignificant when compared to in-band redundancy. In the `course-docs` workload, dedup-lbfs-128 detects just 1,01% more (out-of-band) redundant contents than dedupFS. The `student-projects` workload exhibits more out-of-band redundancy, but still at insignificant levels: dedup-lbfs-128 detects 4,37% more redundant data.

It is certainly easy to identify scenarios where out-of-band redundancy is substantial, as we discuss in Section 6.5. Nevertheless, the results from the above two real concurrent workloads are very meaningful. They capture collaborative scenarios that are very representative of the ones that the present thesis mainly addresses. Furthermore, in both workloads there are numerous evident sources of out-of-band redundancy, which Section 8.3.1.2 enumerates; in fact, our results show that out-of-band redundancy does occur. However, our results confirm that almost all redundancy that we can exploit when synchronizing the new contents at each divergent replica results from contents copied from the initial version, d_1; in other words, in-band redundancy.

The above small advantage in detected redundant data volume of dedup-lbfs over dedupFS does not eliminate the overall advantage of the latter over the former; either in terms of transferred volume or synchronization performance. In order to synchronize the concurrent replicas of either `course-docs` or `student-projects`, every dedup-lbfs variant transfers more bytes and takes more time than any dedupFS variant. More concretely, the best variant of dedup-lbfs (dedup-lbfs-8K) transfers 2,9% more volume the most network-efficient dedupFS variant, dedupFS-128, and takes 29,8% more time than the fastest dedupFS variant, dedupFS-2K. With `student-projects`, in spite of the higher out-of-band redundancy, dedup-lbfs-8K still transfers 1,1% more than dedupFS-128, and is 8,6% slower than dedupFS-2K.

8.3.4.2 Space Requirements of Version Log

A second condition to dedupFS's effectiveness is that the version logs at each synchronizing site span for a sufficiently large time period. Since dedupFS can only detect redundancy across the versions to send and the intersection of the version logs of both sites, the larger such an intersection is, the more redundancy dedupFS may exploit. In the worst case, where both logs do not intersect at all, dedupFS can no longer exploit any redundancy that may exist across both sites.

Figure 8.31 illustrates such a condition with an example of two sites, each holding a divergent version of concurrent workload. As long as both sites are able to maintain sufficiently long logs (in this case, as long as both logs include the old version d_1), dedupFS will be able to exploit redundant chunks c_1 and c_2. However, excessive log truncation can strongly hamper dedupFS's ability of detecting in-band redundancy across both sites, even when such redundancy exists. In the example, truncating the oldest version in the log of at one of the sites will result in an empty intersection between both sites' logs, hence disabling dedupFS from

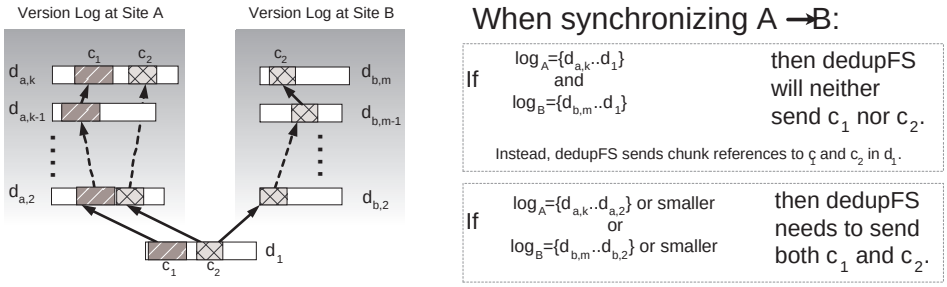


Figure 8.31: Example of impact of log size in redundancy detection in dedupFS. Two sites, *A* and *B*, hold divergent replicas of a concurrent workload (for simplicity, the figure depicts each workload version as a single file). Both divergent replicas originate from a common, initial version, d_1. When *B* synchronizes with *A* (i.e. synchronization from *A* to *B*), dedupFS's ability to exploit redundancy depends on the depth of the logs that each site holds in that moment. Firstly, if *B*'s log no longer holds d_1 (due to log truncation), then chunk c_1 is not redundant across *A* and *B*, thus *A* must transfer its contents. Secondly, such a situation (*B*'s log without d_1) also means that dedupFS will not be able to detect chunk c_2 as redundant across both sites, since both logs do not intersect; although c_2 is, in fact, redundant across both sites.

detecting any in-band redundancy across both sites.

Evidently, it is key that, as a site's log incorporates newer versions, its space overhead remains at acceptable levels, so as to delay version truncation to as late as possible. This section studies such space requirements. We analyze the two collaborative workloads, `course-docs` and `student-projects`. However, we do not consider the same two-version workloads as in the previous sections. Instead, we have obtained daily CVS snapshots of each such workload[19].

We consider sequences of multiple workload versions, $d_1, d_2, .., d_{n-1}, d_n$. Each sequence spans across the whole duration of each workload: in `course-docs`, $n =$ 168 (days), while $n = 74$ (days) in `student-projects`. Each workload version d_i exclusively includes the files whose contents have been modified relatively to the previous day's version, d_{i-1}.

In this experiment, we place ourselves at the last day of the workload, starting with a single version log (holding d_n only). We then add $d_{n-1}, d_{n-2}, ...$ to the log and measure how much additional space each deeper log requires. We consider different schemes for log storage: plain storage, redundancy compression and partial pruning, with different expected chunk sizes.

Figures 8.32 and 8.33 present the results of such an experiment. A first observation is that, even with plain log storage, space cost grows moderately with log depth. With `course-docs`, the space overhead of maintaining the entire 6-month history of daily collaborative activity implies storing less than the size of

[19] `course-docs` and `student-projects` are concurrent workloads. In this section, we consider one of the divergent branches only, for each workload. More precisely, we consider daily versions from d_1 to d_a. The results for the d_b branch yield comparable conclusions.

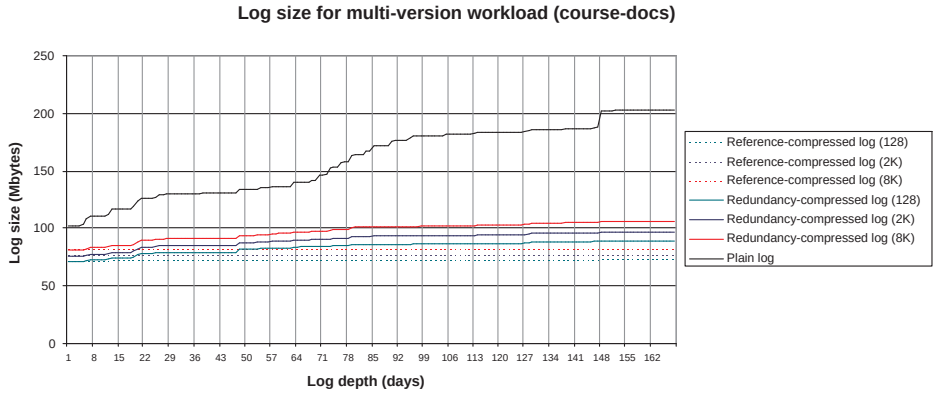


Figure 8.32: Space requirements of version log for multi-version `course-docs` workload.

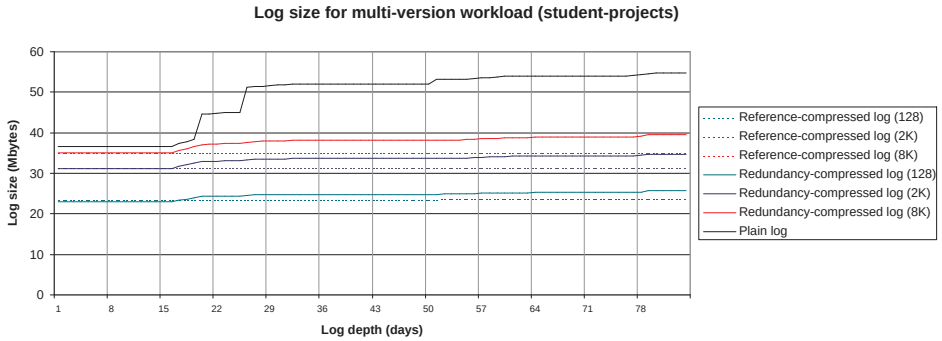


Figure 8.33: Space requirements of version log for multi-version `student-projects` workload.

the current version. More precisely, maintaining the most recent version requires 102 MB, while storing the entire history of 168 versions requires 98% more space. With `student-projects`, maintaining the entire 3-month version history requires 49% more space than the 37 MB of the most recent version. With most devices, for which secondary memory is cheap and abundant, such overheads are usually acceptable.

Furthermore, the more efficient log storage schemes are still able to substantially reduce such a space overhead. Redundancy compression allows logging the entire history of versions with a 25%-31% space overhead in `course-docs`, depending on the expected chunk size (128 B and 8 KB produce the lowest and highest overhead, respectively). In `student-projects`, such an overhead drops to 11.1%-12.6%. Such low overheads are clearly important when we consider memory-constrained devices, such as most mobile ones. Achieving such low overheads per day of logging means that even memory-constrained devices will be able

to maintain knowledge of significantly long version histories.

Finally, partial pruning offers almost negligible overhead. However, as Section 6.3 explains, exploiting the redundancy footprints at the receiver side is not necessarily advantageous for synchronization performance. Hence, we restrict our analysis the case of sites that always act as senders, as Section 6.3 explains. With this scheme, a sender-only site may store the redundancy footprints of the entire histories of `course-docs` and `student-projects` at the cost of 1,94%-0,08% and 2.06%-0.09%, respectively; as Section 6.3 explains, such footprints hold sufficient information to completely replace the whole versions when the site acts as sender during synchronization. It is worth noting that, in contrast to redundancy compression, partial pruning is less efficient with smaller expected chunk sizes, as more chunk references exist.

8.3.5 Local Similarity Detection

In contrast to a synchronization session, the local similarity detection algorithm runs in background. Hence, its performance is neither the central concern of dedupFS's current implementation nor of the present evaluation. Nevertheless, despite not being a critical process, it is still desirable that local similarity detection consumes as less resources as possible (e.g. battery and CPU) during the shortest time possible. Therefore, this section devotes some attention on the performance of the local similarity detection algorithm of dedupFS.

During the experiments in Section 8.3.2 we gathered some measures concerning local similarity detection. We obtained such measures both for the local similarity detection algorithms dedupFS and dedup-lbfs[20].

Recall that dedupFS performs a byte-by-byte comparison between any pair of chunks that it finds with similar hash values. On one hand, this step (which dedup-lbfs discards) imposes a strong performance penalty on dedupFS, as we shall analyze next. On the other hand, the reliance on byte-by-byte comparison makes it possible for dedupFS to use a faster, non-cryptographic hash function (see Section 6.4). Consequently, dedupFS is faster in determining hash values, but may have to byte-by-byte-compare more chunks, due to more frequent hash collisions. If comparing one byte per redundant byte found has a significant performance overhead, hash collisions may amplify it by causing multiple unnecessary byte-by-byte comparisons when false positives occur.

One first relevant measure is, for each byte that dedupFS found redundant, the number of bytes that, on average, dedupFS compares against such a byte. From Table 8.10, we can see that such a measure varies considerably across workloads. While `emacs` and `linux` remain close to the optimal one byte compared per redundant byte, `course-docs` and `usrbin` require more than 2.5 comparisons per

[20] In the case of dedup-lbfs, we refer to the local similarity detection that dedup-lbfs performs for exploiting chunk redundancy in locally stored contents. Obviously, we are not considering the distributed similarity detection algorithm, used for synchronization.

Workload	Bytes compared per redundant byte		
	128 B	2 KB	8 KB
course-docs	2.5	1.1	1.0
student-projects	48.0	21.6	13.3
gcc	1.2	1.8	1.5
emacs	1.0	1.0	1.0
linux	1.0	1.0	1.0
usrbin	2.7	1.1	1.0

Table 8.10: Number of byte-by-byte comparisons per each byte found redundant by dedupFS, for different expected chunk sizes.

Workload	Running Time (s)			Rel. Performance (vs. dedup-lbfs)		
	128 B	2 KB	8 KB	128 B	2 KB	8 KB
course-docs	201.9	30.5	13.2	783%	313%	137%
student-projects	5,484.1	443.7	61.5	25,220%	9,497%	1,229%
gcc	440.0	157.4	95.2	596%	287%	219%
emacs	57.0	15.6	7.7	441%	170%	96%
linux	342.8	118.8	62.5	521%	234%	134%
usrbin	1,056.8	142.8	61.0	706%	307%	152%

Table 8.11: Local similarity detection performance of dedupFS, both absolute and relatively to dedup-lbfs, for different expected chunk sizes.

byte. Surprisingly, `student-projects` exhibits a highly pathological case of frequent hash collisions, reaching 13 to 48 byte comparisons per byte, depending on whether expected chunk size is 8 KB or 128 B, respectively.

Inevitably, the byte-by-byte comparison and the impact of hash collisions has a considerable performance penalty on dedupFS's local similarity detection step. To quantify such a penalty, we compare dedupFS's local similarity detection time with dedup-lbfs's, as we show in Table 8.11. Clearly, dedupFS takes substantially more time to detect local redundancy than dedup-lbfs. Such a performance gap increases as we decrease chunk sizes. On average over all workloads (excluding `student-projects`'s pathological case), dedupFS takes 48% more time with an expected chunk size of 8 KB; decreasing such a parameter to 2 KB and 128 B causes dedupFS to become 2.6 times and 6.1 times slower than dedup-lbfs, respectively. We mainly attribute such a performance decrease to the higher number of chunks, which are directly proportional to the number of disk accesses and chunk hash table lookups during the algorithm execution.

Table 8.12 presents the overall throughput values of dedupFS's local similarity detection step for each workload. Despite the unfavorable relative performance, the throughput of dedupFS's local similarity detection achieves ranges from 746 KB (with 128 B expected chunk size) to 6,600 KB (with 8 KB expected chunk size) of new content handled per second, on average over all workloads (including the problematic `student-projects`).

Workload	Throughput (KBps)		
	128 B	2 KB	8 KB
course-docs	555	3,668	8,515
student-projects	10	119	858
gcc	713	1,994	3,298
emacs	1,768	6,436	13,067
linux	928	2,678	5,087
usrbin	507	3,750	8,775

Table 8.12: Throughput of local similarity detection in dedupFS. Throughput is given by the volume of new content that the similarity detection step can handle per second. Throughput is given in KB per second, for different expected chunk sizes.

In typical interactive collaborative applications, we expect such a throughput range to be higher than the typical rates at which users input new contents and synchronize. Hence, we consider the measured throughput acceptable, though subject to improvement. The use of fast, yet less collision-prone hash functions, and the use of massively available stream processor hardware such as graphics processing units (GPUs) to boost chunk local similarity detection (as proposed in [AKGSN+08]) are certainly interesting directions for future improvement.

8.4 Summary

Departing from an experimental results, we validate the improvements that the contributions of the present thesis attain.

Using a simulator of an optimistic replication system running in a mobile network, we have analyzed the performance and efficiency of the VVWV protocol in relation to other representative commitment protocols. Namely, Primary and Deno. VVWV is substantially more efficient than Deno, particularly as one considers weaker connectivity and/or more realistic update models with narrower sets of active replicas. Such an advantage is visible both in terms of the percentage of updates that commit, and the delays that each replica imposes to users and applications before deciding, and possibly committing, each update. Concerning Primary, VVWV achieves comparable commit ratios. In terms of commitment delays, Primary only outperforms VVWV in scenarios of high contention scenarios due to high numbers of active replicas.

We further evaluate VVWV by considering its agreement decoupling by consistency packet exchange. We confirm that every benefit that we predicted does exist, namely in terms of commitment acceleration, abort minimization and more efficient update propagation. Most importantly, we show that each such benefit may be significant provided that enough non-replicas surround the replicated system and contribute to consistency packet exchange. In general, we observe that, in a system of five replicas, 20 collaborating non-replicas produce visible improvements, which grow to very relevant levels with 30 and 40 non-replicas.

A fundamental conclusion is that, if one considers an alternative solution where non-replicas carry potentially large packets of both update-data and meta-data (e.g. off-line anti-entropy), then a sufficiently large number of resource-constrained non-replicas carrying only lightweight consistency packets may provide comparable or better improvements than the former. This observation is especially meaningful in the context of ubiquitous computing and sensor networks, where resource-constrained nodes are the norm.

We then evaluate dedupFS, a prototypal implementation of our data deduplication solution for efficient replica storage and synchronization. Using dedupFS and other state-of-the-art solutions from each relevant approach to data deduplication, we have replayed real workloads covering a broad set of collaborative activities.

The obtained results complement previous evidence [HH05] that, to some extent, contradicts a general conviction that is subjacent to most literature proposing data deduplication through compare-by-hash [MCM01, CN02, JDT05, ELW+07, BJD06]. Our results show that, for the workloads that the latter works consider and evaluate, there does exist a non-probabilistic solution that performs at least as well as compare-by-hash. Even if applications are willing to tolerate the non-null possibility of data corruption due to hash collisions.

In a first experiment, we consider similar workloads as those that motivate and serve to evaluate recent related work on data deduplication. Such workloads are sequential, hence do not allow out-of-band redundancy. Our results show that, with very few exceptions, dedupFS detects more redundant contents than relevant compare-by-hash solutions (rsync, TAPER and LBFS), while keeping meta-data overhead at low levels. In result, on average over all workloads, the volume that dedupFS transfers across the network during synchronization is substantially lower than any compare-by-hash solution that we consider. Accordingly, such an advantage is also reflected on synchronization performance.

With respect to synchronization schemes that rely of delta encoding, our results show that, for general workloads, dedupFS is significantly more efficient. However, for workloads with a sufficiently stable directory/file tree and where in-object redundancy dominates cross-object redundancy, delta encoding may substantially outperform dedupFS. Hence, an interesting direction for future improvement of dedupFS is to complement dedupFS's similarity detection algorithm and reference encoding with a variant based on delta encoding. Such a hybrid approach is clearly compatible with dedupFS.

Finally, we assert the above promising results under more realistic assumptions. We do that by presenting strong experimental evidence that: (i) out-of-band redundancy does occur rarely in computer-supported collaborative scenarios; and (ii) the space requirements of maintaining sufficiently long version logs are, in general, acceptable even in the environments that the present thesis addresses. We support the first statement with an analysis of two concurrent workloads, taken from real users that actively made use of shared CVS repositories. In both workloads, after 3-month and 6-month periods, the amount of out-of-band redundancy accounts to less than 5% of the total redundancy that one may find in the new contents that each

workload has received. Furthermore, we back up the second claim by showing that, in the previous workloads, dedupFS can maintain the entire version history of 3 and 6 months, respectively, at the acceptable cost of approximately 4% space overhead per logged month of update activity.

Chapter 9

Conclusions

Optimistic replication is a fundamental technique for supporting collaborative work practices in a fault-tolerant manner in weakly connected network environments. As collaboration through weakly connected networks becomes popular (e.g. by using asynchronous groupware applications, or distributed file or database systems, and collaborative wikis), the importance of this technique increases. Examples of such weakly connected environments range from the Internet to ubiquitous computing and pervasive computing environments.

Despite the success of optimistic replication in very specific applications, such as DNS, USENET or electronic mail, it is still hardly applicable to general case collaborative applications. When compared to its pessimistic counterpart, optimistic replication introduces crucial obstacles to its collaborative users. Namely, the temporal distance that separates tentative write operations from their actual commitment with strong consistency guarantees; the inherent possibility of lost work due to aborts; and the increased storage and network requirements that maintaining large version logs imposes are substantial obstacles. Ironically, the very network environments that call for optimistic replication, as a means of dealing with their inherent weak connectivity, tend to substantially amplify all the above obstacles; mostly, due to their weak connectivity and resource constraints.

This thesis targets the three fundamental axes that we identify above: (i) faster strong consistency (ii) leading to less aborted work, while (iii) minimizing both the amount of data exchanged between and stored at replicas. Departing from an initial base, representative of most relevant state-of-the-art optimistically replicated systems, we propose novel improvements that help improve the base system in all three fundamental axes. Furthermore, the positive effect of such improvements grows substantially as we consider environments of weaker connectivity and/or poor device and network resources.

We sum up our central contributions as follows:

1. We formalize epidemic quorum systems and provide a novel, generic characterization of their availability and performance. Our contribution highlights previously undocumented trade-offs and allows an analytical comparison of

proposed epidemic quorum systems.

2. We introduce an update commitment protocol based on weighted voting and version vectors. It provides better fault-tolerance than primary commit schemes while ensuring comparable performance.

3. We describe how to improve update commitment by exploitation of other computational resources, co-existent with the replicated system, as carriers of lightweight consistency meta-data. We show that such a technique may accelerate commitment, avoid aborts, and allow more network-efficient update propagation.

4. We propose a novel update storage and propagation architecture, based on data deduplication, that is at least as efficient as state-of-the-art compare-by-hash approaches and eliminates their crucial shortcoming of being probabilistically error-prone.

We have implemented two prototypes incorporating the contributions: a simulator of distributed environments with poor and partitioned connectivity, and a full-fledged, distributed replicated file system for Linux. Using such prototypes, we thoroughly evaluate our contributions against network and device settings, as well as collaborative application workloads, that are close to the real environments that the thesis targets. The results we obtain support our statement that our contributions can substantially push optimistic replication ahead in all three axes. In other words, our contributions improve users' productivity by supporting more effective data sharing with high availability and performance.

9.1 Future Work

The work we present in this dissertation unveils a number of different and promising directions for future work. In particular, we plan to address the following points, ordered by the location in the thesis that motivates them:

General:

- Support for dynamic replica sets in all contributions of the thesis.
- Weakening of the fault model in order to include Byzantine faults.
- Deployment and global evaluation of a prototype integrating all the contributions of the thesis.

Epidemic Update Commitment Protocols:

- Theoretical results on: (1) conditions under which epidemic quorum systems are effectively advantageous over classical ones; and (2) which are the optimal epidemic coteries.

- Adaptation of VVWV for use of Vector Sets, possibly combined with Version Stamps or Bounded Version Vectors.
- Better definitions of eligible candidates, providing more satisfactory semantic extension of VVWV.

Decoupled Commitment by Consistency Packet Exchange:

- Study and evaluation of effective Consistency Packet buffer management and Consistency Packet substitution algorithms.
- Deployment and evaluation of replicated systems with decoupled commitment by consistency packet exchange with multiple objects.

Versioning-Based Deduplication:

- Hybrid local redundancy algorithms and data structures, which combine version-to-version delta-encoding and general cross-file and cross-version compare-by-hash-based redundancy detection.
- Improved algorithm for detection of recursive redundancy in dedupFS.
- Techniques to eliminate sources of out-of-band redundancy in dedupFS; for instance, techniques to allow transparent version tracking of contents that are uploaded and downloaded from http servers, or exchanged by e-email.
- Extensions of dedupFS to support a distributed backup service and to support efficient job transference in grid computing systems [Sto07].
- Optimization of dedupFS's local similarity detection algorithm, through the use of fast, yet less collision-prone hash functions, and the use of massively available stream processor hardware such as graphics processing units (GPUs) [AKGSN+08].

Bibliography

[3GP] 3GPP. 3rd Generation Partnership Project. http://www.3gpp.org/.

[AAB04] José Bacelar Almeida, Paulo Sérgio Almeida, and Carlos Baquero. Bounded version vectors. In Rachid Guerraoui, editor, *Proceedings of DISC 2004: 18th International Symposium on Distributed Computing*, number 3274 in LNCS, pages 102–116. Springer Verlag, 2004.

[AAS97] D. Agrawal, A. El Abbadi, and R. C. Steinke. Epidemic algorithms in replicated databases (extended abstract). In *PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 161–172, New York, NY, USA, 1997. ACM.

[AB98] Ericsson AB. Edge - introduction of high-speed data in gsm/gprs networks. http://www.ericsson.com/technology/whitepapers/, 1998.

[AD76] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the International Conference on Software Engineering*, 1976.

[AFM05] Siddhartha Annapureddy, Michael J. Freedman, and David Mazieres. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, USA, May 2005.

[AGJA06] R. C. Agarwal, K. Gupta, S. Jain, and S. Amalapurapu. An approximation to the greedy algorithm for differential compression. *IBM J. Res. Dev.*, 50(1):149–166, 2006.

[AKGSN+08] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan, and Matei Ripeanu. Storegpu: exploiting graphics processing units to accelerate distributed storage systems. In Manish Parashar, Karsten Schwan, Jon B. Weissman, and Domenico Laforenza, editors, *HPDC*, pages 165–174. ACM, 2008.

[AW96] Yair Amir and Avishai Wool. Evaluating quorum systems over the internet. In *Symposium on Fault-Tolerant Computing*, pages 26–35, 1996.

[Bar03] João Barreto. Information sharing in mobile networks: a survey on replication strategies. Technical Report RT/015/03, Inesc-ID Lisboa, 2003.

[BDG+06] N Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[BF04] João Barreto and Paulo Ferreira. A replicated file system for resource constrained mobile devices. In *Proceedings of IADIS International Conference on Applied Computing*, 2004.

[BF05a] João Barreto and Paulo Ferreira. A Highly Available Replicated File System for Resource-Constrained Windows CE .Net Devices. In *3rd International Conference on .NET Technologies*, May 2005.

[BF05b] João Barreto and Paulo Ferreira. An efficient and fault-tolerant update commitment protocol for weakly connected replicas. In *Euro-Par 2005*, Springer Verlag Lecture Notes in Computer Science, pages 1059–1068. Springer Verlag, 2005.

[BF05c] João Barreto and Paulo Ferreira. Efficient file storage using content-based indexing. In *20th ACM Symposium on Operating Systems Principles (Poster Session)*, ACM Digital Archive. ACM, October 2005.

[BF07a] J. Barreto and P. Ferreira. The availability and performance of epidemic quorum algorithms. Technical Report RT/10/2007, INESC-ID, February 2007.

[BF07b] J. Barreto and P. Ferreira. Version vector weighted voting protocol: efficient and fault-tolerant commitment for weakly connected replicas. *Concurrency and Computation: Practice and Experience*, 19(17):2271–2283, 2007.

[BF07c] João Barreto and Paulo Ferreira. Understanding epidemic quorum systems. In *EuroSys 2007 (Poster Session)*, March 2007.

[BF08a] João Barreto and Paulo Ferreira. Efficient corruption-free duplicate elimination in distributed storage systems. Technical Report 49, INESC-ID, September 2008.

[BF08b] João Barreto and Paulo Ferreira. The obscure nature of epidemic quorum systems. In *ACM HotMobile 2008: The Ninth Workshop on Mobile Computing Systems and Applications*, Napa Valley, CA, USA, February 2008. ACM Press.

[BFS07] J. Barreto, P. Ferreira, and M. Shapiro. Exploiting our computational surroundings for better mobile collaboration. In *8th International Conference on Mobile Data Management (MDM 2007)*, pages 110–117. IEEE, May 2007.

[BG83] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[BGL+06] Roberto Baldoni, Rachid Guerraoui, Ron R. Levy, Vivien Quéma, and Sara Tucci Piergiovanni. Unconscious Eventual Consistency with Gossips. In *Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006)*, 2006.

[BI03] M. Boulkenafed and V. Issarny. Adhocfs: Sharing files in wlans. In *Proceeding of the 2nd IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, USA, April 2003.

[BJD06] Deepak R. Bobbarjung, Suresh Jagannathan, and Cezary Dubnicki. Improving duplicate elimination in storage systems. *Transactions on Storage*, 2(4):424–448, 2006.

[BLNS82] Andrew Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Commun. ACM*, 25(4):260–274, 1982.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BvR94] K. Birman and R. van Renesse, editors. *Reliable Distributed Computing With the ISIS Toolkit*. Number ISBN 0-8186-5342-6. IEEE CS Press, March 1994.

[Byr99] Randy Byrne. *Building Applications with Microsoft Outlook 2000 Technical Reference*. Microsoft Press, Redmond, WA, USA, 1999.

[C+93] Per Cederqvist et al. Version management with CVS. [Online Manual] `http://www.cvshome.org/docs/manual/`, as of 03.09.2002, 1993.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *FIDET '74: Proceedings of the*

1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control, pages 249–264, New York, NY, USA, 1974. ACM Press.

[CDN+96] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical internet object cache. In *USENIX Annual Technical Conference*, pages 153–164, 1996.

[Cho06] Yung Chou. Get into the Groove: Solutions for Secure and Dynamic Collaboration. http://technet.microsoft.com/en-us/magazine/cc160900.aspx, 2006.

[CK02] Ugur Cetintemel and Pete Keleher. Light-weight currency management mechanisms in mobile and weakly-connected environments. *The Journal of Distributed and Parallel Databases (JDPD)*, 11:53–71, 2002.

[CKBF03] U. Cetintemel, P. J. Keleher, B. Bhattacharjee, and M. J. Franklin. Deno: A decentralized, peer-to-peer object replication system for mobile and weakly-connected environments. *IEEE Transactions on Computer Systems (TOCS)*, 52, July 2003.

[cKF01] Ugur Çetintemel, Peter J. Keleher, and Michael J. Franklin. Support for speculative update propagation and mobility in deno. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 509–516, 2001.

[CM99] S. Corson and J. Macker. Mobile ad hoc networking (MANET): Routing protocol performance issues and evaluation considerations. Internet Request for Comment RFC 2501, Internet Engineering Task Force, January 1999.

[CMP04] Brian W. Fitzpatrick C. Michael Pilato, Ben Collins-Sussman. *Version Control with Subversion*. O'Reilly, 2004.

[CN01] Landon P. Cox and Brian D. Noble. Fast Reconciliations in Fluid Replication. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 449–458, 2001.

[CN02] L. Cox and B. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of Fifth USENIX Symposium on Operating Systems Design and Implementation*, December 2002.

[Con96] United States Congress. Health insurance portability and accountability act (hipaa). http://www.hipaa.com, 1996.

[Con02] United States Congress. Sarbanes-Oxley Act (SOX), Public Law 107–204, 107th Congress, 2002.

[CS99] Peter H. Carstensen and Kjeld Schmidt. Computer supported cooperative work: New challenges to systems design. In Kenji Itoh, editor, *Handbook of Human Factors*, pages 619–636. Asakura Publishing, 1999. in Japanese, English Version available from http://www.itu.dk/people/schmidt/publ.html.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43:225–267, 1996.

[CVS04] Tae-Young Chang, Aravind Velayutham, and Raghupathy Sivakumar. Mimic: raw activity shipping for file synchronization in mobile file systems. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 165–176. ACM Press, 2004.

[CW96] Ing-Ray Chen and Ding-Chau Wang. Analyzing dynamic voting using petri nets. In *SRDS '96: Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS '96)*, page 44, Washington, DC, USA, 1996. IEEE Computer Society.

[CW08] File Compression and Decompression (Windows). Microsoft developer network. http://msdn.microsoft.com/en-us/library/aa364219.aspx, 2008.

[DBTW94] K. Petersen M. J. Spreitzer M. M. Theimer D. B. Terry, A. J. Demers and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings Third International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, Texas, September 1994.

[DCGN03] Michael Dahlin, Bharat Baddepudi V. Chandra, Lei Gao, and Amol Nayate. End-to-end wan service availability. *IEEE/ACM Transactions on Networking*, 11(2):300–313, 2003.

[Deu50] L. Deutsch. Zlib compressed data format specification version 3. Internet Request for Comment RFC 1950, Internet Engineering Task Force, 1950.

[DGH+87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, New York, NY, USA, 1987. ACM Press.

[DGMS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys*, 17(3):341–370, 1985.

[DPS+94] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–7, Santa Cruz, California, 8-9 1994.

[ELW+07] Kave Eshghi, Mark Lillibridge, Lawrence Wilcock, Guillaume Belrose, and Rycharde Hawkes. Jumbo store: providing efficient incremental upload and versioning for a utility rendering service. In *FAST'07: Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies*, pages 22–22, Berkeley, CA, USA, 2007. USENIX Association.

[ET05] Kave Eshghi and Hsiu Tang. A framework for analyzing and improving content-based chunking algorithms. Technical Report 2005-30, Hewlett-Packard Labs, 2005.

[FB99] Armando Fox and Eric A. Brewer. Harvest, yield, and scalable tolerant systems. In *HOTOS '99: Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, page 174, Washington, DC, USA, 1999. IEEE Computer Society.

[FG00] Svend Frølund and Rachid Guerraoui. X-ability: a theory of replication. In *PODC*, pages 229–237, 2000.

[FGL+96] Alan Fekete, David Gupta, Victor Luchangco, Nancy A. Lynch, and Alexander A. Shvartsman. Eventually-serializable data services. In *Symposium on Principles of Distributed Computing*, pages 300–309, 1996.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol http/1.1. Internet Request for Comment RFC 2616, Internet Engineering Task Force, 1999.

[Fid91] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[FZ94] George H. Forman and John Zahorjan. The challenges of mobile computing. *Computer*, 27(4):38–47, 1994.

[GCK01] Jean Dollimore George Coulouris and Tim Kindberg. *Distributed Systems: Concepts and Design*. Pearson Education 2001, 3 edition, 2001.

[GHM+90] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus Replicated File System. In *Proc. 1990 Summer USENIX Conf.*, Anaheim, June 11-15 1990.

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating System Principles SOSP 7*, pages 150–162, Asilomar Conference Grounds, Pacific Grove CA, 1979. ACM, New York.

[Gol93] Richard Golding. Modeling replica divergence in a weak-consistency protocol for global-scale distributed data bases. Technical Report UCSC-CRL-93-09, UC Santa Cruz, February 1993.

[GRR+98] Richard G. Guy, Peter L. Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald J. Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *ER Workshops*, pages 254–265, 1998.

[HAA02] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. Partial database replication using epidemic communication. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 485, Washington, DC, USA, 2002. IEEE Computer Society.

[Hen03] Val Henson. An analysis of compare-by-hash. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

[HG02] Val Henson and Jeff Garzik. Bitkeeper for kernel developers. `http://infohost.nmt.edu/~val/ols/bk.ps.gz`, 2002.

[HH05] Val Henson and Richard Henderson. Guidelines for using compare-by-hash. `http://infohost.nmt.edu/~val/review/hash2.html`, 2005.

[HHH+98] James J. Hunt, James J. Hunt, James J. Hunt, James J. Hunt, Kiem-Phong Vo, Kiem-Phong Vo, Kiem-Phong Vo, Kiem-Phong Vo, Walter F. Tichy, Walter F. Tichy, Walter F. Tichy, and Walter F. Tichy. Delta algorithms: an empirical analysis. *ACM Trans. Softw. Eng. Methodol.*, 7(2):192–214, April 1998.

[HSAA03] JoAnne Holliday, Robert Steinke, Divyakant Agrawal, and Amr El Abbadi. Epidemic algorithms for replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1218–1238, 2003.

[IEE97] IEEE. IEEE 802.11 Wireless Local Area Networks Working Group. `http://grouper.ieee.org/groups/802/11/index.html`, 1997.

[IK01] K. Ingols and I. Keidar. Availability study of dynamic voting algorithms. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 247, Washington, DC, USA, 2001. IEEE Computer Society.

[IN06] Claudia-Lavinia Ignat and Moira C. Norrie. Draw-Together: Graphical editor for collaborative drawing. In *Int. Conf. on Computer-Supported Cooperative Work (CSCW)*, pages 269–278, Banff, Alberta, Canada, November 2006.

[IOM+07] Claudia-Lavinia Ignat, Gérald Oster, Pascal Molli, Michèlle Cart, Jean Ferrié, Anne-Marie Kermarrec, Pierre Sutra, Marc Shapiro, Lamia Benmouffok, Jean-Michel Busca, and Rachid Guerraoui. A comparison of optimistic approaches to collaborative editing of wiki pages. In *CollaborateCom*, pages 474–483. IEEE, 2007.

[JDT05] N. Jain, M. Dahlin, and R. Tewari. Taper: Tiered approach for eliminating redundancy in replica sychronization. In *USENIX Conference onf File and Storage Technologies (FAST05)*, Dec 2005.

[JHE99] Jin Jing, Abdelsalam Sumi Helal, and Ahmed Elmagarmid. Client-server computing in mobile environments. *ACM Computing Surveys*, 31(2):117–157, 1999.

[JI01] Udo Fritzke Jr. and Philippe Ingels. Transactions on partially replicated data based on reliable and atomic multicasts. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 284–291, 2001.

[JM90] S. Jajodia and David Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, 15(2):230–280, 1990.

[JM05] Flavio Paiva Junqueira and Keith Marzullo. Coterie availability in sites. In *DISC*, pages 3–17, 2005.

[JMR97] H. V. Jagadish, Inderpal Singh Mumick, and Michael Rabinovich. Scalable versioning in distributed databases with commuting updates. In *ICDE '97: Proceedings of the Thirteenth International Conference on Data Engineering*, pages 520–531, Washington, DC, USA, 1997. IEEE Computer Society.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[Kel99] P. Keleher. Decentralized replicated-object protocols. In *Proc. of the 18th Annual ACM Symp. on Principles of Distributed Computing (PODC'99)*, 1999.

[KRSD01] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. In *20th Symp. on Principles of Dist. Comp. (PODC)*, Newport RI (USA), August 2001. ACM SIGACT-SIGOPS.

[KS91] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–25. ACM SIGOPS, October 1991.

[KS93] Puneet Kumar and Mahadev Satyanarayanan. Log-based directory resolution in the Coda file system. In *Proc. Second Int. Conf. on Parallel and Distributed Information Systems*, pages 202–213, San Diego, CA (USA), 1993.

[KWK03] Brent ByungHoon Kang, Robert Wilensky, and John Kubiatowicz. Hash history approach for reconciling mutual inconsistency in optimistic replication. In *23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03)*, 2003.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690–691, September 1979.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[Lam05] Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[LC01] B. Leuf and W. Cunningham. *The wiki way: Quick collaboration on the web*. Addison-Wesley, 2001.

[LH86] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 229–239. ACM Press, 1986.

[LH87] Debra A. Lelewer and Daniel S. Hirschberg. Data compression. *ACM Comput. Surv.*, 19(3):261–296, 1987.

[LKBH+88] Jr. Leonard Kawell, Steven Beckhardt, Timothy Halvorsen, Raymond Ozzie, and Irene Greif. Replicated document management in a group communication system. In *CSCW '88: Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, page 395, New York, NY, USA, 1988. ACM Press.

[LLS99] Yui-Wah Lee, Kwong-Sak Leung, and Mahadev Satyanarayanan. Operation-based update propagation in a mobile file system. In *USENIX Annual Technical Conference, General Track*, pages 43–56. USENIX, 1999.

[LLSG92] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391, 1992.

[LOM94] K. Lidl, J. Osborne, and J. Malcolm. Drinking from the firehose: Multicast usenet news. In *Proc. of the Winter 1994 USENIX Conference*, pages 33–45, San Francisco, CA, 1994.

[LS90] Eliezer Levy and Abrahram Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4):321–374, December 1990.

[LSP95] Lamport, Shostak, and Pease. The byzantine generals problem. In *Advances in Ultra-Dependable Distributed Systems, N. Suri, C. J. Walter, and M. M. Hugue (Eds.), IEEE Computer Society Press*. 1995.

[Maca] J. MacDonald. Versioned file archiving, compression, and distribution. Available via `http://citeseer.ist.psu.edu/250029.html`.

[Macb] J. MacDonald. xdelta. http://code.google.com/p/xdelta/.

[Mac00] J. MacDonald. File system support for delta compression. Masters thesis, University of California at Berkeley, 2000.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[Maz00] David Mazieres. *Self-certifying file system*. PhD thesis, 2000. Supervisor-M. Frans Kaashoek.

[MCM01] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001.

[MD95] Paul V. Mockapetris and Kevin J. Dunlap. Development of the domain name system. *SIGCOMM Comput. Commun. Rev.*, 25(1):112–122, 1995.

[Mer79] Ralph Charles Merkle. *Secrecy, authentication, and public key systems*. PhD thesis, Stanford, CA, USA, 1979.

[Mic00] Microsoft. *Windows 2000 Server: Distributed systems guide*, chapter 6, pages 299–340. Microsoft Press, 2000.

[MM85] Webb Miller and Eugene W. Myers. A file comparison program. *Software - Practice and Experience (SPE)*, 15(11):1025–1040, 1985.

[MNP07] Dahlia Malkhi, Lev Novik, and Chris Purcell. P2P replica synchronization with vector sets. *SIGOPS Oper. Syst. Rev.*, 41(2):68–74, 2007.

[MSC+86] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith. Andrew: a distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, 1986.

[MSP+08] Stefan Miltchev, Jonathan M. Smith, Vassilis Prevelakis, Angelos Keromytis, and Sotiris Ioannidis. Decentralized access control in distributed file systems. *ACM Comput. Surv.*, 40(3):1–30, 2008.

[MT05] Dahlia Malkhi and Douglas B. Terry. Concise version vectors in winfs. In Pierre Fraigniaud, editor, *DISC*, volume 3724 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2005.

[MW82] T. Minoura and G. Wiederhold. Resilient extended true-copy token scheme for a distributed database systems. *IEEE Transactions on Software Engineering*, SE-8:173–189, 1982.

[Nat95] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard*. National Institute for Standards and Technology, Gaithersburg, MD, USA, April 1995. Supersedes FIPS PUB 180 1993 May 11.

[Now89] Bill Nowicki. Nfs: Network file system protocol specification. Internet Request for Comment RFC 1094, Internet Engineering Task Force, March 1989.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[PBAB07] Zachary N. J. Peterson, Randal Burns, Giuseppe Ateniese, and Stephen Bono. Design and implementation of verifiable audit trails for a versioning file system. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.

[Ped01] Fernando Pedone. Boosting system performance with optimistic distributed protocols. *Computer*, 34(12):80–86, 2001.

[PGS03] Fernando Pedone, Rachid Guerraoui, and André Schiper. The Database State Machine Approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[PL88] Jehan-François Pâris and Darrell D. E. Long. Efficient dynamic voting algorithms. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 268–275, Washington, DC, USA, 1988. IEEE Computer Society.

[PL91] Calton Pu and Avraham Leff. Replica control in distributed systems: as asynchronous approach. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 377–386, New York, NY, USA, 1991. ACM.

[POS90] *System Application Program Interface (API) [C Language]*. Information technology—Portable Operating System Interface (POSIX). IEEE, 1990.

[PP04] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *ATEC'04: Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

[PPR+83] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detec-

tion of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, 1983.

[PS02] Fernando Pedone and André Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing Journal*, 15(2):97–107, 2002.

[PSM03] N. Preguiça, Marc Shapiro, and Caroline Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *CoopIS'2003 The Eleventh International Conference on Cooperative Information Systems*, number 2888 in Lecture Notes in Computer Science, pages 38–55. Springer-Verlag, 11 2003.

[PST+97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating SystemsPrinciples (SOSP-16), Saint Malo, France*, 1997.

[PSTT96] Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. Bayou: Replicated database services for world-wide applications. In *7th ACM SIGOPS European Workshop*, Connemara, Ireland, 1996.

[PSYC03] Justin Mazzola Paluska, David Saff, Tom Yeh, and Kathryn Chen. Footloose: A case for physical eventual consistency and selective conflict resolution. In *5th IEEE Workshop on Mobile Computing Systems and Applications*, pages 170–180, Monterey, CA, USA, October 9–10, 2003.

[PW95] David Peleg and Avishai Wool. The availability of quorum systems. *Information and Computation*, 123(2):210–223, 1995.

[QD02] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, Monterey,CA, 2002.

[Rab81] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[Rat98] David Howard Ratner. Roam: A Scalable Replication System for Mobile and Distributed Computing. PhD Thesis 970044, University of California, 31, 1998.

[RC01] Norman Ramsey and El″od Csirmaz. An algebraic approach to file synchronization. *SIGSOFT Softw. Eng. Notes*, 26(5):175–185, 2001.

[RD01] Antony I. T. Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, pages 188–201, 2001.

[RO92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[RPCC93] Peter Reiher, Gerald Popek, Jeff Cook, and Stephen Crocker. Truffles—a secure service for widespread file sharing. In *PSRG Workshop on Network and Distributed System Security*, 1993.

[RRP99] David Ratner, Peter Reiher, and Gerald Popek. Roam: A scalable replication system for mobile computing. In *DEXA '99: Proceedings of the 10th International Workshop on Database & Expert Systems Applications*, page 96, Washington, DC, USA, 1999. IEEE Computer Society.

[RRPK01] David Ratner, Peter L. Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. Replication requirements in mobile environments. *Mobile Networks and Applications*, 6(6):525–533, 2001.

[RS96] Michel Raynal and Mukesh Singhal. Logical time: Capturing causality in distributed systems. *Computer*, 29(2):49–56, 1996.

[RT99] E. Royer and Chai-Keong Toh. A review of current routing protocols for ad hoc mobile wireless networks. *Personal Communications, IEEE [see also IEEE Wireless Communications]*, 6(2):46–55, 1999.

[Sat02] M. Satyanarayanan. The evolution of coda. *ACM Transactions on Computer Systems (TOCS)*, 20(2):85–124, 2002.

[SB04] Marc Shapiro and Karthik Bhargavan. The Actions-Constraints approach to replication: Definitions and proofs. Technical Report MSR-TR-2004-14, Microsoft Research, March 2004.

[SBK04] Marc Shapiro, Karthikeyan Bhargavan, and Nishith Krishna. A constraint-based formalism for consistency in replicated systems. In *Proc. 8th Int. Conf. on Principles of Dist. Sys. (OPODIS)*, number 3544 in Lecture Notes In Computer Science, pages 331–345, Grenoble, France, December 2004.

[SBL99] Yasushi Saito, Brian N. Bershad, and Henry M. Levy. Manageability, availability and performance in porcupine: a highly scalable, cluster-based mail service. In *SOSP '99: Proceedings of the*

17th ACM Symposium on Operating Systems Principles, pages 1–15, New York, NY, USA, 1999. ACM Press.

[SBS06] Pierre Sutra, João Barreto, and Marc Shapiro. An asynchronous, decentralised commitment protocol for semantic optimistic replication. Rapport de recherche 6069, December 2006.

[SBS07] Pierre Sutra, João Barreto, and Marc Shapiro. Decentralised commitment for optimistic semantic replication. In *15th International Conference on Cooperative Information Systems (CoopIS 2007)*, Vilamoura, Algarve, Portugal, November 2007.

[SCF02] Paulo Sergio, Almeida Carlos, and Baquero Victor Fonte. Version stamps - decentralized version vectors. In *Proc. of the 22nd International Conference on Distributed Computing Systems*, 2002.

[Sch95] Bruce Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[SE98] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Conf. on Comp.-Supported Cooperative Work (CSCW)*, page 59, Seattle WA, USA, November 1998.

[SFH+99] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 110–123, New York, NY, USA, 1999. ACM Press.

[SGMV08] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. Pergamum: replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–16, Berkeley, CA, USA, 2008. USENIX Association.

[SGS+00] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised system. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 12–12, Berkeley, CA, USA, 2000. USENIX Association.

[SGSG02] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in a comprehensive versioning file system. In *In Proceedings of USENIX Conference on File and Storage Technologies*, 2002.

[Sha05] Y. Shafranovich. Common format and mime type for comma-separated values (csv) files. Internet Request for Comment RFC 4180, Internet Engineering Task Force, October 2005.

[SHPF04] Tara Small, Zygmunt J. Haas, Alejandro Purgue, and Kurt Fristup. *Handbook of Sensor Networks*, chapter 11, A Sensor Network for Biological Data Acquisition. CRC Press, July 2004.

[SKKM02] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. *SIGOPS Operating Systems Review*, 36(SI):15–30, 2002.

[SKW+02] Edward Swierk, Emre Kiciman, Nathan C. Williams, Takashi Fukushima, Hideki Yoshida, Vince Laviano, and Mary Baker. The Roma personal metadata service. *Mobile Networks and Applications*, 7(5):407–418, 2002.

[SM94] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[Sri95] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. Internet Request for Comment RFC 1832, Internet Engineering Task Force, 1995.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[SS08] Pierre Sutra and Marc Shapiro. Fault-tolerant partial replication in large-scale database systems. In *Europar*, pages 404–413, Las Palmas de Gran Canaria, Spain, August 2008.

[SSB06] Pierre Sutra, Marc Shapiro, and João Barreto. An asynchronous, decentralised commitment protocol for semantic optimistic replication. Research Report 6069, INRIA, 12 2006.

[SSP06] Nicolas Schiper, Rodrigo Schmidt, and Fernando Pedone. Optimistic algorithms for partial database replication. In Alexander A. Shvartsman, editor, *OPODIS*, volume 4305 of *Lecture Notes in Computer Science*, pages 81–93. Springer, 2006.

[Sto79] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, SE-5:188–194, 1979.

[Sto07] Heinz Stockinger. Defining the grid: a snapshot on the current view. *J. Supercomput.*, 42(1):3–17, 2007.

[SV02] Jonathan S. Shapiro and John Vanderburgh. Cpcms: A configuration management system based on cryptographic names. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 207–220, Berkeley, CA, USA, 2002. USENIX Association.

[SVF07] Nuno Santos, Luis Veiga, and Paulo Ferreira. Vector-field consistency for ad-hoc gaming. In *ACM/IFIP/Usenix International Middleware Conference (Middleware 2007)*, Lecture Notes in Computer Science. Springer, September 2007.

[SW00] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM*, August 2000.

[TKS+03] Niraj Tolia, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, Adrian Perrig, and Thomas Bressoud. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 127–140, San Antonio, TX, June 2003.

[TM98] A. Trigdell and P. Mackerras. The rsync algorithm. Technical report, Australian National University, 1998. http://rsync.samba.org.

[TRA99] Francisco J. Torres-Rojas and Mustaque Ahamad. Plausible clocks: constant size logical clocks for distributed systems. *Distrib. Comput.*, 12(4):179–195, 1999.

[TTC+90] Gomer Thomas, Glenn R. Thompson, Chin-Wan Chung, Edward Barkmeyer, Fred Carter, Marjorie Templeton, Stephen Fox, and Berl Hartman. Heterogeneous distributed database systems for production use. *ACM Comput. Surv.*, 22(3):237–266, 1990.

[TTP+95] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, pages 172–182. ACM Press, 1995.

[Val93] Céline Valot. Characterizing the accuracy of distributed timestamps. *SIGPLAN Not.*, 28(12):43–52, 1993.

[Vog99] Werner Vogels. File system usage in Windows NT 4.0. In *SOSP '99: Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 93–109, New York, NY, USA, 1999. ACM Press.

[WB84] Gene T.J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 233–242, New York, NY, USA, 1984. ACM Press.

[WC97] D. Wessels and K. Claffy. Internet cache protocol. Internet Request for Comment RFC 2186, Internet Engineering Task Force, 1997.

[Wei91] M. Weiser. The computer for the twenty-first century. *Scientific American*, 265:94–104, 1991.

[Wil91] P. Wilson. *Computer Supported Cooperative Work: An Introduction*. Oxford, Intellect Books, 1991.

[WPE+83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The locus distributed operating system. In *In Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 49–70, 1983.

[WPS+00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274, Taipei, Taiwan, R.O.C., April 2000. IEEE Computer Society Technical Commitee on Distributed Processing.

[YK04] Lawrence You and Christos Karamanolis. Evaluation of efficient archival storage techniques. In *Proceedings of 21st IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies*, 2004.

[YV00] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of Operating Systems Design and Implementation*, pages 305–318, 2000.

[YV01] Haifeng Yu and Amin Vahdat. The costs and limits of availability for replicated services. In *Symposium on Operating Systems Principles*, pages 29–42, 2001.

[ZFS08] OpenSolaris Community: ZFS. Zfs. http://www.opensolaris.org/os/community/zfs/, 2008.

[ZL78] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[ZPS00] Y. Zhang, V. Paxson, and S. Shenker. The stationarity of internet path properties: Routing, loss, and throughput. *ACIRI Technical Report*, 2000.

Appendix A

Update Pruning in the Initial Solution

This section complements the initial solution from Section 3.2.6 with an update pruning scheme that bounds the memory requirements of the optimistic replication service. The following solution is closely based on Bayou's pruning scheme [PST+97], with minor differences. Besides some differences in terminology, we introduce an optimization to the update propagation when update truncation is supported. In Bayou, the condition for transference of the whole base value ignores the tentative updates that the receiver replica may hold. In some cases, false positives may occur, and the sender replica transmits the whole base value unnecessarily. More precisely, if the sender replica has committed some update u and truncated it (hence applied it upon its base value) and the receiver replica has u in its log, but still in a tentative form, then Bayou decides to (unnecessarily) send the base value. Our algorithm avoids such false positive decisions.

In order to offer its committed and tentative values, in spite of update pruning, each replica, a, maintains a *base value*, denoted $base_a$, and an *update log*, denoted log_a. Both implement the conceptual sch_a; from both, we may produce the committed and tentative values that result from sch_a.

Pruning must, however, follow the log order (which is the same as r-schedule order). Furthermore, only stable updates may be pruned; tentative updates, instead, cannot be pruned since they must, at any moment, be individually available to rollback or to be reordered. Hence, conceptually, there exists a prefix of $stable_a$ that comprises the that some replica a has pruned so far; we denote such a schedule as $pruned_a$.

Similarly to $stable_a$, $pruned_a$ converges to a common, totally ordered schedule, as more stable updates become pruned (due to totally-ordered EEC). Similarly to the r-schedule, each replica a maintains a version vector representing $pruned_a$, which we denote $\lceil pruned_a \rceil$. When some replica prunes an update from the log, it applies it upon the base value, in case the update was scheduled as executable; and

it updates $[pruned_a]$ to reflect the new pruned update.[1] In addition, each replica a maintains an integer, OSN_a (using the terminology in [PST+97]), which counts the number of updates the replica has pruned so far.

From the base value and log, a replica a may obtain its tentative and committed values. The former results from the ordered execution of $\underline{log_a}$ upon $base_a$; while the latter results from the ordered execution of $\underline{(stable_a \cap log_a)}$ upon $base_a$ (i.e. the execution of the committed prefix of log_a upon $\overline{base_a}$).

By allowing update pruning, it may no longer be possible to bring two replicas up-to-date by propagating individual updates from each one's log. In fact, some cases will require transmitting the whole base value. Hence, we need to extend the update propagation protocol so as to support update truncation, as in Algorithm 23.

Algorithm 23 Update propagation protocol from site A to site B, with support for update truncation.

```
1:  for all Object x that B replicates, with replica b do
2:    B sends id_x, [sch_b] and OSN_{r_B} to A.
3:    if A replicates x, with replica a then
4:      if [pruned_a] ⋠ [sch_b] then
5:        A sends base_a, [pruned_a] and OSN_a to B
6:        B sets base_b ← base_a
7:        B sets [pruned_b] ← [pruned_a]
8:        B sets OSN_b ← OSN_a
9:        B sets CSN_b ← OSN_a
10:       B sets [sch_b] ← merge([sch_b], [pruned_a])
11:       for all u in log_b such that [pruned_b][u.rid] ≥ u.lid do
12:         B discards u from log_b
13:     Let u be the first update in sch_a.
14:     while u exists do
15:       if [sch_b][u.sid] < u.lid then
16:         A sends u to B.
17:         B sets sch_b ← scheduleUpdate(u, sch_b)
18:         B sets [sch_b][u.sid] ← u.lid
19:       Let u be the next update in sch_a.
```

The extended update propagation protocol starts by checking whether it needs to transfer the base value of the sender replica (line 4). Such a condition is true if $[sch_b]$ does not dominate $[pruned_a]$, which implies that the receiver replica's r-schedule at least does not hold one of the pruned updates at the sender replica. In this case, the only way to transmit the effect of such an update is to send the whole base value, in spite of the expectedly higher network requirements.

When B receives the base value, it overwrites the previous $base_b$ with the

[1] Conceptually, the base value of some replica r results from the ordered application of $\underline{pruned_a}$. The replica's log, in turn, is a suffix of sch_a containing the updates in sch_a that are not in $\overline{pruned_a}$.

sender's base value (line 6). Accordingly, B updates $[pruned_b]$ to the version vector representing the received base value (line 7). Since the new base value includes new stable values, then it necessarily embeds a larger $stable_b$ than the previous one; accordingly, B updates CSN_b with the number of updates that the new base value embeds, given by OSN_a (line 9). Further, b updates the $[sch_b]$ version vector so that it includes such new updates (line 10). Finally, b discards any updates in log_b that have already been applied upon the new base value (lines 11–11).

Appendix B

On Epidemic Quorum Systems

B.1 Comparison with Classical Quorum Systems and Discussion

The availability of classical quorum systems is extensively studied [JM90, CW96, IK01, PW95, JM05]. However, such results are not valid for epidemic quorum systems, as both approaches differ radically in the way sites obtain quorums. In the classical approach, a coordinator (typically the site proposing the value) runs a synchronous atomic commit (e.g. two-phase commit) to obtain a quorum to atomically vote for a value. The epidemic approach collects votes in an asynchronous coordinator-free way, by means of pair-wise epidemic propagation of vote information. Furthermore, in the classical approach, the coordinator decides whether to proceed with the agreement (if it has obtained a quorum of voters), or to withdraw the votes (due to unavailable quorum). The epidemic approach does not allow vote withdrawal before agreement, and agreement is a local decision of each site, based on the locally collected vote information.

In consequence, ECs, in contrast to classical coteries, may take into account, not only the set of voters for a given value (the *quorum*), but also the set of voters for the competing values (the *anti-quorums*); plurality-based weighted voting [Kel99] is an example of a coterie that the classical approach does not allow.

One may compare classical and epidemic quorum systems by trying to answer two questions: (a) can one build an epidemic quorum system from a classical one (and vice-versa)?; and (b) given a classical and an epidemic quorum system, which one is better?

The first question has already been partially answered by Holliday et al. in [HSAA03]. They propose epidemic quorum systems constructed from classical coteries [PW95]. A classical coterie is a set of sets of sites, $Q^1,..Q^m$, such that, for all Q^i and Q^j, $Q^i \cap Q^j \neq \emptyset$ (intersection property) and $\forall Q^j \in QSet(\mathcal{E}_{cl}), Q^i \not\subseteq Q^j$ (minimality property). We formalize their idea and prove it safe as follows.

Definition 23. *EC equivalent of classical coterie*

Let $\mathcal{C}$ be a classical coterie. Its EC equivalent, denoted $\mathcal{E}_{\mathcal{C}}$, is a set of q-vote configurations such that $QSet(\mathcal{E}_{\mathcal{C}}) = \mathcal{C}$ and $AQSet(\mathcal{E}_{\mathcal{C}}) = \emptyset$.

Proposition 1. *Let $\mathcal{C}$ be a classical coterie. Its EC equivalent, $\mathcal{E}_{\mathcal{C}}$, is an EC.*

The $\mathcal{E}_{Maj}$ EC introduced in Section 4.1.1.2 is an example of an EC equivalent of a classical coterie.

Since the universe of ECs is larger than the universe of EC equivalents, one cannot always build a classical quorum system from an epidemic one. An example is the $\mathcal{LP}$ EC.

Concerning the second question, we may try to compare our results with the results available for classical quorum systems. Consider an EC, $\mathcal{E}$, and a classical coterie, $\mathcal{C}$. Further, assume $\mathcal{M}_p$.

The probability of a quorum of $\mathcal{C}$ being accessible to p, $q_{\mathcal{C}}$, is given by

$$q_{\mathcal{C}} = 1 - \sum_{i=0}^{|Sites|} a_i^{\mathcal{C}} f^i (1-f)^{y-i}$$

where $(a_0^{\mathcal{C}}, .., a_{|Sites|}^{\mathcal{C}})$ is the availability profile of $\mathcal{C}$ (as defined in [PW95]), and f denotes the probability of each site either being faulty, or correct but not in the same partition as p. Using our model, f is given by $p_f + (1-p_f)h$.

Hence, the probability of a quorum being accessible to p within r rounds, $q_{\mathcal{C}}^r$ is given by the sum of a geometric series:

$$q_{\mathcal{C}}^r = \sum_{x=0}^{r-1} (1-q_{\mathcal{C}})^x q_{\mathcal{C}} = \frac{q_{\mathcal{C}}(1-(1-q_{\mathcal{C}})^r)}{q_{\mathcal{C}}}$$

$q_{\mathcal{C}}$ is an upper bound on the effective availability of $\mathcal{C}$, as it ignores contention and faults that may occur during the quorum obtention site.[1] Further, $q_{\mathcal{C}}^r$ is an upper bound on the probability that a classical quorum system decides within d rounds, $P_d^{\mathcal{C}}$. Still, it enables us to determine a sufficient (not necessary) condition for $\mathcal{E}$ to outperform $\mathcal{C}$:

$$P_d^{\mathcal{E}}(r) > q_{\mathcal{C}}^r, \text{ for all } r > 0.$$

The above condition is a partial answer to the second question. A more precise definition of $q_{\mathcal{C}}$, which takes into account the effects of contention and faults during quorum obtention, would produce a more satisfactory comparison of classical and epidemic quorum systems. This is out of the scope of the thesis and a goal for future work.

B.2 Complementary Proofs and Definitions

Proposition 1. Let qc and qd be q-vote configurations such that $qd \succ qc$. For any A_i^{qd}, either there exists A_j^{qc} such that $[A_j^{qc}]_{qc} \supseteq [A_i^{qd}]_{qd}$, or $[\emptyset]_{qc} \supseteq [A_i^{qd}]_{qd}$.

[1] As Ingols and Keidar note and experimentally study in [IK01].

Proof. Clearly, $\bigcup_i A_c^i \subseteq \bigcup_j A_d^j$, which implies $[\emptyset]_c \supseteq [\emptyset]_d$ (1). Let f be the injective function such that $\forall_{1 \leq k \leq n_c} : A_k^c \subseteq A_{f(k)}^d$. For each $i \in \{1,..,n_c\}$, $A_i^c \subseteq A_j^d$, thus $A_j^d = A_i^c \cup (A_j^d \backslash A_i^c)$, since $c \subseteq d$. Since d is a q-vote configuration, $A_j^d \cap A_x^d = \emptyset, \forall x \neq j$; which implies $(A_j^d \backslash A_i^c) \cap A_x^d = \emptyset$ (2). Since $\forall A_y^c : y \neq i, \exists A_j^d : A_j^d \supseteq A_y^c$, then it follows from (2) that $(A_j^d \backslash A_i^c) \cap A_y^c = \emptyset$. Thus, $(A_j^d \backslash A_i^c) \subseteq [\emptyset]_c$. Finally, $[A_j^d]_d = A_j^d \cup [\emptyset]_d = A_i^c \cup (A_j^d \backslash A_i^c) \cup [\emptyset]_d$, which implies, by (1) and (2), $[A_j^d]_d \subseteq A_i^c \cup [\emptyset]_c$.

The proof for the case of each A_j^d such that $\nexists i \in \{1,..,n_c\} : f(i) = j$ is direct. Clearly, $A_j^d \cap A_x^c = \emptyset, \forall x \neq j$; so, $A_j^d \subseteq [\emptyset]_c$. Therefore, $[A_j^d]_d = A_j^d \cup [\emptyset]_d \subseteq [\emptyset]_c$. □

Lemma 2. Let A and B be two sites where $e_A = e_B$. Then, $votes_A(x) \subseteq [votes_B(x)]_{c_r}$ for all $x \in Val$.

Let p and r be two processes where $e_A = e_B$. Then, $v_x^p \subseteq [v_x^r]_{c_r}$ for all $x \in Val$.

Proof. Clearly, the algorithm ensures that only the vote of a given process may only be cast by the process itself, and only once each election. Hence, $votes_A(x) \cap votes_B(y) = \emptyset$ for any processes p, r and values x, y. Therefore, $votes_A(x) \backslash votes_B(y) \subseteq [\emptyset]_{c_r}$. By standard set manipulation, we get $[votes_A(x)]_{c_r} = votes_A(x) \cup [\emptyset]_{c_r} \supseteq (votes_A(x) \cap votes_B(y)) \cup [\emptyset]_{c_r} \supseteq (votes_A(x) \cap votes_B(y)) \cup (votes_A(x) \backslash votes_B(y)) = votes_A(x)$. □

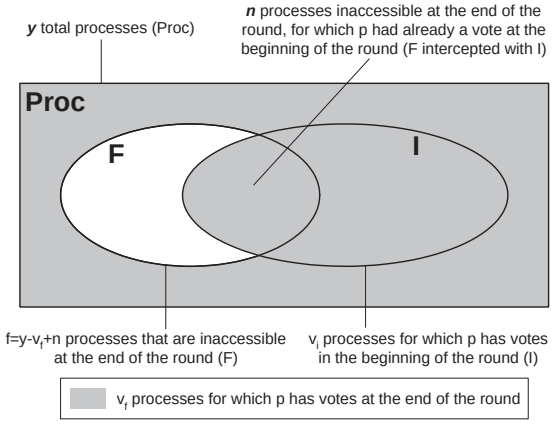


Figure B.1: Event sets when considering a round where p, starting with v_i votes, ends up collecting v_r votes.

Lemma 4. *In $\mathcal{M}_p$, p_v is given by:*

$$p_v(v_i \rightarrow v_f; r) = \begin{cases} \sum_{n=0}^{v_i} \binom{v_i}{n} \binom{|Sites| - v_i}{|Sites| - v_f} h^{(y - v_f + n)} (1-h)^{(v_f - n)}, & \text{if} \quad r = 1 \\ \sum_{v_x = v_i}^{v_f} p_v(v_x \rightarrow v_f; 1) p_v(v_i \rightarrow v_x; r-1), & \text{if} \quad r > 1 \end{cases} \tag{B.1}$$

Proof. Starting with v_i votes, at the end of one round, the number of votes (v_f) is the number of correct sites in the partition of p whose vote was not yet known by p, plus v_i (where n of the voters that have contributed to v_i are inaccessible). As Figure B.1 depicts, the number of inaccessible sites, f, is given by $|Sites| - v_f + n$.

$p_v(v_i \rightarrow v_f; 1)$ is given by the summing the probabilities of collecting v_f votes for each n, ranging from 0 to v_i. For each value n, p collects v_f votes at the end of the round if: (i) n out of the v_i votes is inaccessible, and (ii) out of the sites that started the round without vote ($|Sites| - v_i$), $|Sites| - v_f$ remain in that condition. By simple combinatorial reasoning, the number of such events is given by $\binom{v_i}{n}\binom{|Sites|-v_i}{|Sites|-v_f}$. Further, the probability of each such event is the probability of exactly f sites being inaccessible at the end of the round; i.e. $h^{(y-v_f+n)}(1-h)^{(v_f-n)}$.

The case of $r > 1$ is simpler. At a first round, p may collect v_x votes (where $v_i \leq v_x \leq v_f$) (with probability $p_v(v_x \rightarrow v_f; 1)$) and collect the votes still missing in the remaining rounds (with probability $p_v(v_i \rightarrow v_x; r-1)$). Clearly, $p_v(v_i \rightarrow v_f; r)$ is given by the sum of the probabilities for each v_x. □

Theorem 2. The availability of EC $\mathcal{E}$ is given by:

$$\sum_{n=0}^{|Sites|} \frac{\binom{y}{n} (1-p_f)^n \; p_f^{(|Sites|-n)} dec_{\mathcal{E}}(n)}{1 - rep_{\mathcal{E}}(n)} \tag{B.2}$$

Proof. Assuming n correct sites, the probability of eventual decision is the probability of any possible sequence of consecutive *repeats*, followed by one *decide*. Since such sequences are exclusive events, their probability is the following sum: $dec_{\mathcal{E}}(n) + dec_{\mathcal{E}}(n) rep_{\mathcal{E}}(n) + dec_{\mathcal{E}}(n) rep_{\mathcal{E}}(n)^2 + ..$, where each i-th term denotes the probability that the first $i-1$ elections have outcome *repeat* and, finally, i-th election decides. The previous formula is a sum of a geometric series, which is known to the same as $dec_{\mathcal{E}}(n) \sum_{i=0}^{\infty} rep_{\mathcal{E}}(n)^{\;i} = \frac{dec_{\mathcal{E}}(n)}{1-rep_{\mathcal{E}}(n)}$. The probability of exactly n correct sites out of y is $\binom{y}{n}(1-p_f)^n \; p_f^{(|Sites|-n)}$. Again, the events of exactly n (with $0 \leq n \leq y$) sites are exclusive; hence, it follows directly that the probability for any n and any decision sequence is given by (B.2). □

Theorem 3. Probability of decision within r rounds

The probability that an arbitrary site, A, starting with $V_p = \emptyset$, decides any value within r or less rounds, $P_d^{\mathcal{E}}(r)$, is given by:

$$P_d^{\mathcal{E}}(r) = \begin{cases} 0, & \text{if} \quad r = 0 \\ \sum_{v_r=0}^{|Sites|} p_v(0 \rightarrow v_r; r) dec_{\mathcal{E}}(v_r) + f(0,r), & \text{if} \quad r > 0 \end{cases} \tag{B.3}$$

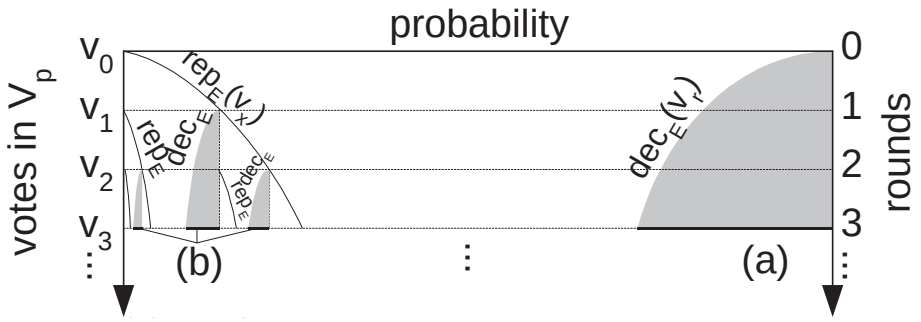


Figure B.2: Illustration of the probabilities that comprise $P_d^{\mathcal{E}}(r)$, up to 3 rounds. Decision cases are greyed out, and correspond to components (a) and (b).

where f is defined as:

$$f(v_i,r) = \sum_{v_x=v_i}^{|Sites|} p_v(v_i \rightarrow v_x;1)[(rep_{\mathcal{E}}(v_x) - rep_{\mathcal{E}}(v_i))P_d^{\mathcal{E}}(r-1) + f(v_x,r-1)] \tag{B.4}$$

Proof. $P_d^{\mathcal{E}}(0)$ is zero since *decide-condition* is impossible with an empty V_p. With $r > 0$, one may distinguish two situations that fulfill *decide-condition*, (a) and (b); Figure B.2 depicts such situations. (a) corresponds to the cases where, after r or less rounds, p collected enough votes, v_r, to decide in a single election. The probability of (a) is the sum, for each v_r, of the probability of collecting exactly v_r votes *and* deciding with such an amount of votes; i.e., $\sum_{v_r=0}^{|Sites|} p_v(0 \rightarrow v_r;r) dec_{\mathcal{E}}(v_r)$.

The cases where at least one *repeat-condition* is verified, and followed by a *decide*, correspond to (b). We will show that, at the start of a given round, having v_i initial votes, the probability of deciding, including at least one *repeat*, within the next r or less rounds, is given by $f(v_i,r)$. In such a round, at the end of which $v_x \geq v_i$ votes are collected, $f(v_i,r)$ is given by the sum of the probabilities of two exclusive events (multiplied by the probability of collecting v_x votes, $p_v(v_i \rightarrow v_x;1)$, for each $v_r \geq v_i$): (i) either v_x produce the *repeat* outcome, and so a new election starts; or (ii) no such outcome occurs. Hence, (i) happens if, at the end of the round, p determines *repeat without previously having obtained such outcome with v_i votes* (probability of $rep_{\mathcal{E}}(v_x) - rep_{\mathcal{E}}(v_i)$) and decides in the remaining elections ($P_d^{\mathcal{E}}(r-1)$). Otherwise (ii), a *decide* after at least one *repeat* may still occur, starting with the new v_r votes, and within the remaining rounds $(r-1)$ (with probability $f(v_x,r-1)$).

Clearly, (b) is obtained by $f(0,r)$. $(a)+(b)$ gives the probability $P_d^{\mathcal{E}}(r)$ when $r > 0$. □

Theorem 1. Let $\mathcal{C}$ be a classical coterie. Its EC equivalent, $\mathcal{E}_{\mathcal{C}}$, is an EC.

Proof. By Def. 6, and since has empty anti-quorums, $\mathcal{E}_C$ is an EC if, $\forall qc, qd \in \mathcal{E}_C$, (1) $\forall j : [A_j^{qc}]_{qc} \not\supseteq Q^{qd}$, (2) $[\emptyset]_{qc} \not\supseteq Q^{qd}$, and (3)$qc \not\subseteq qd$ and $qd \not\subseteq qc$. (1) is vacuously true. (3) is ensured by the minimality property. To prove (2), assume, for the purpose of contradiction, that there exists $qd \in \mathcal{E}_C$ such that $[\emptyset]_{qc} \supseteq Q^{qd}$. By the intersection property, $\exists X \neq \emptyset : Q^{qc} \cap Q^{qd} = X$. Therefore, $X \subseteq Q^{qd} \subseteq [\emptyset]_{qc}$ (hence, $X \subseteq [\emptyset]_{qc}$), and $X \subseteq Q^{qc}$. So, $X \subseteq Q^{qc} \cap [\emptyset]_{qc}$. Thus $Q^{qc} \cap [\emptyset]_{qc} \neq \emptyset$; however, it follows directly from Def. 3 that $Q^{qc} \cap [\emptyset]_{qc} = \emptyset$, which is a contradiction. □