Universidade do Minho
Escola de Engenharia
Departamento de Informática

Semantically Reliable Group Communication

por

José Orlando Roque Nascimento Pereira

Dissertação apresentada à Universidade do Minho para
obtenção do grau de Doutor em Informática

Orientador:
Luís Eduardo Teixeira Rodrigues
(Professor Associado, FC/Univ. Lisboa)

Co-orientador:
Rui Carlos Mendes de Oliveira
(Professor Auxiliar)

Braga
Outubro de 2002

Este trabalho foi parcialmente financiado pela Fundação para a Ciência e a Tecnologia através do projecto SHIFT (POSI/32869/CHS/2000).

Resumo

A utilização de computadores e redes de transmisssão de dados em diversas aplicações do quotidiano, torna desejável a adopção de técnicas de tolerância a faltas em sistemas baseados em hardware e software não especializados. A comunicação em grupo é, neste contexto, uma tecnologia particularmente atraente, pois oferece ao programador garantias de fiabilidade que simplificam significativamente a aplicação de técnicas de tolerância a faltas.

No entanto, a experiência tem mostrado que a concretização deste modelo em sistemas heterogéneos e de grande escala levanta problemas de desempenho. Embora as limitações de desempenho possam ser evitadas através de um relaxamento das garantias de fiabilidade, os protocolos resultantes são normalmente menos úteis, nomeadamente, na replicação com coerência forte. O desafio reside pois no relaxamento das garantias de fiabilidade sem deixar de oferecer um modelo adequado à programação de aplicações tolerantes a faltas.

Esta dissertação estuda modelos e mecanismos que permitem conciliar as vantagens da comunicação em grupo com o elevado desempenho, recorrendo para isso ao enfraquecimento selectivo das garantias oferecidas pelos protocolos. A nosso proposta consiste no uso pelo protocolo de informação sobre a semântica das mensagens, por forma a escolher quais delas têm que ser fiavelmente transmitidas, daí a fiabilidade semântica. Em diversas aplicações, algumas mensagens revogam ou transmitem implicitamente outras mensagens enviadas recentemente, tornando-as obsoletas durante a sua transmissão. Ao omitir apenas as mensagens obsoletas, o desempenho pode ser melhorado sem impacto na correcção da aplicação.

São apresentados as especificações e os algoritmos de um conjunto protocolos de comunicação em grupo com fiabilidade semântica, incluindo ordenação e sincronismo virtual. Os protocolos são então avaliados com um modelo analítico, um modelo de simulação e um protótipo. A discussão de uma aplicação concreta ilustra a interface de programação e o desempenho resultante.

Abstract

Current usage of computers and data communication networks for a variety of daily tasks, calls for widespread deployment of fault tolerance techniques with inexpensive off-the-shelf hardware and software. Group communication is in this context a particularly appealing technology, as it provides to the application programmer reliability guarantees that highly simplify many fault tolerance techniques.

It has however been reported that the performance of group communication toolkits in large and heterogeneous systems is frequently disappointing. Although this can be overcome by relaxing reliability guarantees, the resulting protocol is often much less useful than group communication, in particular, for strong consistent replication. The challenge is thus to relax reliability and still provide a convenient set of guarantees for fault tolerant programming.

This thesis addresses models and mechanisms that by selectively relaxing reliability guarantees, offer both the convenience of group communication for fault tolerant programming and high performance. The key to our proposal is to use knowledge about the semantics of messages exchanged to determine which messages need to be reliably delivered, hence semantic reliability. In many applications, some messages implicitly convey or overwrite other messages sent recently before, making them obsolete while still in transit. By omitting only the delivery of obsolete messages, performance can be improved without impact on the correctness of the application.

Specifications and algorithms for a complete semantically reliable group communication protocol suite are introduced, encompassing ordered and view synchronous multicast. The protocols are then evaluated with analytical and simulation models and with a prototype implementation. The discussion of a concrete application illustrates the resulting programming interface and performance.

Acknowledgements

I want to thank my adviser, Luís Rodrigues, for his insightful guidance, his constant encouragement, and his patient and prompt replies to my suggestions and scribblings, despite the distance. I also want to thank Rui Oliveira and Francisco Moura for their support and for providing me the unusual but interesting possibility of working at the University of Minho with an external adviser.

I am also grateful to the other members of the committee for their helpful comments and sugestions in improving the clarity and correctness of the final text. Namely, to Prof. R. Guerraoui (E.P.F. Lausanne), Dr. K. Guo (AT&T Bell Labs), Prof. M. Mota (U. Minho), Prof. J. Silva (U. Coimbra), and Prof. J. Valença (U. Minho).

I thank the people of the Distributed Systems Group, specially Rui Oliveira and António Sousa, for many lengthy and useful technical discussions. I also thank the researchers in Shift and Rumor projects, which have been hard at work implementing and testing semantically reliable protocols. I want to thank all that have read and commented the papers which contain most of the material of this thesis.

Finally, I want to thank to the Departamento de Informática, for hosting me, and to FCT, for supporting the work on semantically reliable protocols through projects Shift and Rumor.

Contents

List of Figures

Chapter 1

Introduction

The reliance on computers for monitoring and controlling critical processes, from medical equipment to avionics and air traffic control, has long been a reality and has justified the research and development of fault tolerant computer systems. The criticality of the functions performed justifies expensive custom hardware components and the involvement of skilled developers.

Current usage of computers and data communication networks in a variety of daily tasks brings the issue of fault tolerance techniques to a wider audience: Even if no human life is at stake, outages can have significant economic impact. For instance, it has been estimated that an hour of downtime of ebay.com servers results in a loss of $225,000 [Pat02]. Such applications drive the seek for cost effective solutions for high performance fault tolerant computing.

Faults can be tolerated by replicating system components [Cri91]. When a component is affected, a replica is used to ensure the continuation of the service. The major issue in managing replication is coordinating replicas both during normal operation and across component failures. The goal is to ensure that the service provided by the set of replicated servers is indistinguishable from the service provided by a single server [HW90].

A cost effective approach to replication is to use off-the-shelf components,

both for servers and interconnecting networks, and use standard software packages to coordinate replication. By leveraging commodity hardware, the system designer has maximum flexibility to configure the system for the desired performance level. Resorting to off-the-shelf software packages reduces the need for developer skills in fault tolerance that can be concentrated on the application itself.

An appropriate tool for the implementation of software replication is group communication software [Pow96]. Group communication offers a message passing interface and encompasses reliable multicast and management of current group membership, alleviating the programmer from the burden of tracking the operational status of each system component. Message passing is a programming paradigm that is already familiar to distributed system programmers. What makes group communication a solid foundation for fault tolerant systems is that, in contrast to other message passing middleware, it provides strong guarantees about message delivery despite faults. This hides most of the complexity of programming a fault tolerant distributed system and the result is that intuitive solutions to replication problems are often correct.

Low cost and the high performance of off-the-shelf computer systems make them attractive for high throughput services. On the other hand, it can be observed that computer systems experience transient performance perturbations. These occur in disk subsystems, scheduling, virtual memory, interference by background applications and system tasks, and network operation. These happen even in local area networks and in closely controlled environments and seem to be unavoidable given the complexity of current computer systems [ADAD01]. Load peaks can result in congestion phenomena, such as thrashing in virtual memory subsystems [Den68] and packet loss in networks [Jac88]. In such situations, the throughput of those components is reduced below their nominal capacity.

Although the correctness of group communication and its applications is

not affected by such phenomena, the impact in performance can be dramatic. It has been reported that the performance of group communication in real systems is often disappointing [PS97]. Unfortunately, this happens despite improvements in the implementation of group communication that theoretically scales to very large groups and high throughput [Bir99]. In fact, the performance degradation observed in real world conditions derives directly from the interaction of transient performance perturbations and the strong guarantees offered by group communication and on which stands its usefulness. The solution to this is to relax the guarantees offered by group communication sufficiently to overcome performance issues [BHO+99, BPRS98, RBAR00].

1.1 Problem statement

The problem is that the usefulness of group communication in the development of fault tolerant applications rests precisely on those strong guarantees which are an obstacle to performance. Proposals offering weaker guarantees are much harder to use in fault tolerant systems and thus are not a substitute for group communication. The developer is left with a choice between performance and fault tolerance that is incompatible with requirements of emerging distributed applications. The challenge is thus to relax reliability without endangering the convenience of group communication.

1.2 Summary of contributions

This thesis addresses this dilemma between performance and fault tolerance by proposing a novel reliability criterion: *semantic reliability*. The key to our proposal is to use knowledge about the semantics of messages exchanged to determine which messages become obsolete while in transit and can thus be discarded immediately. This suffices to improve performance and requires

little additional effort to ensure the correctness of fault tolerant applications.

Semantic reliability is presented as specifications and algorithms for a complete semantically reliable group communication protocol suite, encompassing ordered and view synchronous multicast. This protocol suite is then applied to fault tolerant information dissemination and strongly consistent replication.

1.3 Results

A prototype of a semantically reliable multicast protocol is presented and evaluated. Along with a simple analytical model and a simulation model, it shows how protocol configuration and application traffic parameters influence the performance of semantic reliability. The prototype implementation illustrates also how semantic reliability interacts with protocol implementation mechanism, and thus, illustrates how existing group communication protocols can be adapted for semantic reliability.

The discussion of applying semantic reliability to a concrete application, specifically, the replication of distributed multi-player game service, illustrates what are the required modifications to application code and shows the performance of semantic reliability in a real application.

1.4 Dissertation outline

Chapter 2 motivates the work by introducing group communication. This includes explaining why group communication is an appropriate tool for developing fault tolerant distributed applications. It also explains what are the challenges of deploying group communication, specifically, the problem of sustaining stable high throughput in real systems.

Chapter 3 presents the intuition underlying semantic reliability: Some mes-

sages become obsolete while still in transit and can be omitted without endangering correctness. It then presents the formalization of the semantics that need to be conveyed to the protocol; the mechanisms that can be used to implement it; and examples how it can be used in typical applications.

Chapter 4 introduces the specifications and algorithms of semantically reliable protocols for group communication. This shows how the semantically reliable group communication compares with standard group communication and which are the challenges in implementing it. This chapter is complemented by Appendix A that gives a more detailed algorithm and correctness proof of a key protocol. Appendix B addresses the issues in implementing the proposed algorithms within typical protocol mechanisms.

Chapter 5 evaluates the impact of semantic reliability in the performance of group communication. This is achieved by introducing and comparing the results of an analytical model, a simulation model and an implementation of semantically reliable multicast.

Chapter 6 explores the application of semantic reliability to distributed multi-player games. This includes characterization of the application traffic and evaluation of the performance gains in a real setting.

To conclude, Chapter 7 summarizes the main contributions of this dissertation and discusses future work directions.

Related publications

Preliminary versions of portions of this dissertation have been published:

[1] J. Pereira, L. Rodrigues, and R. Oliveira. Semantically reliable multicast protocols. In *IEEE International Symposium on Reliable Distributed Systems*, Oct. 2000.

This paper motivates and introduces semantic reliability. The performance of the protocol is estimated with the analytical and simulation models using profiling data from a stock exchange application.

[2] J. Pereira, L. Rodrigues, and R. Oliveira. Semantically reliable multicast: Current status and future work. In *International Symposium on Reliable Distributed Systems (DISC)*, Brief Announcement, Oct. 2000.

This short paper introduces an early definition of semantically reliable multicast.

[3] J. Pereira, L. Rodrigues, and R. Oliveira. Fault-tolerant replication of high throughput services. In *IEEE International Conference on Distributed Systems and Networks*, Student Forum, Jun. 2000.

This short paper introduces a simple protocol for totally ordered semantically reliable multicast, describing its application to strongly consistent replication.

[4] J. Pereira, L. Rodrigues, and R. Oliveira. Reducing the cost of group communication with semantic view synchrony. In *IEEE International Conference on Distributed Systems and Networks*, Jun. 2002.

This paper defines semantic view synchrony and its application to strong consistent replication using the primary-backup approach. It also introduces a representation for the obsolescence relation, both for the programmer interface as well as for protocol internals. The protocol is applied to a multiplayer game.

[5] J. Pereira, L. Rodrigues, and R. Oliveira. Semantically reliable broadcast: Sustaining high throughput in reliable distributed systems. In P. Ezhilchelvan and A. Romanovsky (eds.), *Concurrency in Dependable Computing*, Chapter 10, Kluwer Academic Publishers, 2002. ISBN 1-4020-7043-8.

Liveness properties of semantically reliable protocols their impact in application correctness are addressed in this paper by introducing a formal specification, an algorithm and its correctness proof.

[6] N. Carvalho, J. Pereira, and L. Rodrigues. Concretização de protocolos com fiabilidade semântica. In *Conferência sobre Redes de Computadores*, Sep. 2002.

This paper describes the ongoing implementation of semantically reliable group communication within a modular protocol composition framework.

[7] J. Pereira, L. Rodrigues, and R. Oliveira. Semantically reliable multicast: Definition, implementation and performance evaluation. In *I*EEE Transactions on Computers Special Issue on Reliable Distributed Systems, 2003 (to appear).

This paper summarizes previous research on semantic reliability, describes the specification and discussing performance results obtained with analytical and simulation models, and evaluates a protocol implementation prototype.

Chapter 2

Group Communication

This chapter motivates our work by briefly introducing group communication and the resulting trade-off between ease-of-use and performance. Application scenarios illustrate how strong reliability guarantees greatly simplify the programming of distributed fault tolerant applications. However, experience shows that strong reliability is also a fundamental obstacle to the performance of group communication, namely, to throughput stability.

2.1 Overview

Group communication toolkits support message passing within groups of processes by offering membership management and reliable multicast services. Examples of group communication toolkits are Isis and Horus [BvR94], Ensemble [Hay98], xAMp [RV92], Transis [ADKM92], Newtop [EMS95], Phoenix [Mal96], Spread [ADS00] and Appia [MPR01].

Membership management keeps track of which processes are operational and mutually reachable, taking into account both voluntary requests to join and leave the group as well as process failures and network partitions. By ensuring that a common membership is observed by all participants, many distributed algorithms are simplified.

Reliable multicast can informally be described as ensuring delivery of all messages to all destination processes that do not fail. The current composition of the group serves as an implicit destination set for reliable multicast. Several definitions are possible and differ subtly in their guarantees in the presence of process failures [HT94]. Several message ordering criteria in addition to reliable multicast are provided by group communication [HT94]. Message ordering simplifies application programming by ensuring that each message is handled in a predictable context resulting from previous messages.

Group communication differs from other message passing middleware in the consistency guarantees enforced in face of process and network faults by coordinating membership changes with message delivery. This is known as *view synchrony* [Bir93, SS93a] and can be superficially described as totally ordering message delivery and view changes, thus enabling processes to handle each message in a common membership context. This reduces the complexity of coping with process failure when programming applications [SKM00].

Notice that services such as group membership and view synchronous reliable multicast encapsulate solutions to fundamental problems arising in the development of fault tolerant distributed systems such as consensus [CT96, GS01]. This means that, although apparently similar to best-effort multicast protocols [FJL+97], the resulting programming paradigm is in fact as powerful as distributed atomic transactions [GS95]. This makes group communication suitable for fault tolerant applications such as strong consistent replication of services [GS97a, SS93a] while maintaining a simple and general purpose programming interface.

2.2 Specification

We briefly survey specification of group communication services for reference in the following sections. Although subtly different specifications of each ser-

vice exist, we choose to present only one that is representative, both in the sense that it is provided by most toolkits as well as that it is sufficient for the presentation of the applications in the next section. Where appropriate, alternatives are briefly discussed. Comprehensive surveys of group membership and view synchrony properties are found in [HS95, CKV01]. Reliable multicast properties are found in [HT94, CKV01]. Algorithms implementing this specification can be found in [GS01, SS93b].

2.2.1 System model

The specification of group communication services is presented in the context of an asynchronous message passing system model augmented with a failure detector [CT96]. Briefly,[1] the system is modeled as a set of sequential processes fully connected by a network of point-to-point message passing channels such that:

- Processes can only fail by crashing and do not recover, thus ceasing to send or receive further messages. Byzantine or arbitrary faults [Cri91] are not considered. A process that does not crash is said correct. A majority of processes are assumed to be correct.

- Processes communicate only by exchanging messages, thus excluding any form of shared memory. Channel reliability means that if both the sender and the receiver are correct, messages that are sent are eventually received [BCBT96].

- Asynchrony means that there is no bound on relative execution speeds of processes or on the time that takes a message to be transmitted.

- The failure detector oracle available to each process is of class $\diamond$S, thus making consensus solvable [FLP85, CHT96].

[1] The model is described in detail Chapter 4.

Group communication protocols can also be specified in the context of timed models [CF99]. This enables reasoning about timeliness properties that are guaranteed by some toolkits [RV92]. Nevertheless, using an asynchronous model to describe the correctness of a protocol leads to more general specifications and algorithms, that can later be refined to consider timeliness [OPS01].

2.2.2 Group membership

The current composition of a group is usually referred to as the *view* and a membership change notification as *installing a view*. The i^{th} view is installed by a process p by delivering a special control message v_i^p. Each view v_i^p includes the identification of the view and of the set of processes which constitute the current membership of the group.

Membership protocols differ in how new views are formed [RSB93]. Those supporting the *primary partition* model [BSS91] enforce a total order of view installation events, thus ensuring that each view is uniquely preceded and succeeded by a single other view. Primary partition group membership means that:

Primary Partition: If process p installs view v_i^p and process q installs view v_i^q, then $v_i^p = v_i^q$.

As all processes agree on the i^{th} view, it is possible to refer to v_i^p simply as v_i.

The alternative is the *partitionable* model [ADKM92] allowing installation of concurrent disjoint views. New views result from merging or splitting previous views. This allows continued operation when a majority of processes is not reachable, at the expense of consistency [FB96].

In both situations, the events used to trigger view changes are determinant in the liveness guarantees offered to application programs [LH99]. Examples of triggering mechanisms used in existing toolkits are unreliable failure detectors [Mal96, CKV01], resource availability [CBDS01] and network connectivity [BDM01].

2.2.3 Reliable multicast

The multicast service is used through a pair of primitives: *multicast*(m) and *deliver*(m). A process executing the multicast primitive initiates the transmission of a message and is said to *multicast* m. By executing the deliver primitive, transmission is completed by handing over a message to the application at a destination process, which is said to *deliver* m. The definition of reliable multicast is as follows [HT94]:

Validity: If a correct process multicasts a message m, then eventually it delivers m.

Agreement: If a correct process delivers a message m, then all correct processes eventually deliver m.

Integrity: For every message m, every process delivers m at most once and only if m was previously multicast by some process.

A stronger form of reliable multicast which also enforces agreement in processes that are not correct is provided by some group communication toolkits. The resulting strengthened reliability model is known as Uniform Agreement [HT94] or Safe Delivery [CKV01]:

Uniform Agreement: If a process delivers a message m, then all correct processes eventually deliver m.

Although useful in maintaining consistency in distributed applications [GT91], implementation of Uniform Agreement is more costly. Its usage is therefore avoided when not strictly necessary, for instance, when existing explicit acknowledgment mechanisms at the application level make it redundant.

2.2.4 Order

The simplest criterion for ordering deliveries is First-In-First-Out (FIFO), in which messages from the same sender are delivered in the order they were

multicast [HT94, CKV01]:

Reliable FIFO: If a process multicasts a message m before it multicasts a message m', no process delivers m' without previously delivering m.

A step further is given by ordering delivery according to the causal "happens before" relation [Lam78]. A message m causally precedes a message m' if some process sends m before sending m'; or receives m before sending m'; or some m'' exists such that m precedes m'' and m'' precedes m'. Causal order multicast is [HT94, CKV01]:

Reliable Causal: If a message m causally precedes message m', no process delivers m' without previously delivering m.

This is useful, for instance, when determining a global snapshot of the system [CL85].

Most group communication toolkits also allow to totally order the delivery of concurrent messages [HT94, CT96], resulting in messages being delivered in the same order to all processes:

Uniform Total Order: If a process delivers a message m before it delivers a message m', no process delivers m after delivering m'.

Notice that total order is orthogonal to both FIFO and causal ordering and thus can be combined with both. Total order is more costly to implement as it requires coordinating the delivery of concurrent messages. On the other hand, it is useful in implementing replicated services using the active replication approach [Sch93]. Totally ordered reliable multicast is also often called *atomic multicast*, as the effect of the multicast operation appears to occur instantaneously as it logically does not overlap with the delivery of other messages.

Alternative versions of FIFO and causal ordering do not require previous delivery of all predecessors, but prevent only out-of-order delivery [CKV01]. However, reliable version of ordering properties is more useful and as easy

to implement. Alternative formulations of the total order are *weak* total order [WS95], that is not enforced for processes that are expelled from the group and is thus less costly to implement, and total order to multiple overlapping groups [GS97b, RGS98], that enforces total order among messages with different destination sets.

2.2.5 View synchrony

The coordination of reliable multicast and group membership is provided by *view synchrony*. If a process multicasts (resp. delivers) a message m after installing some view v_i and before installing any other view v_j we say m *is multicast (resp. delivered) in view* v_i. The definition is as follows [CKV01]:

View Synchrony: If a process p installs two consecutive views v_i and v_{i+1} and delivers a message m in view v_i, then all other processes installing both v_i and v_{i+1} deliver m in view v_i.

An alternative formulation of view synchrony applies also to processes being excluded [SS93b]. The rationale of the strengthened definition is similar to that of Uniform Agreement and has identical consequences both in consistency and implementation cost.

In addition to View Synchrony, it is also possible to enforce an additional restriction on message delivery regarding view installation:

Sending View Delivery: If a process p multicasts a message m in view v_i, then p does not deliver m in a view other than v_i.

Enforcing simultaneously Sending View Delivery and and Validity requires that processes are prevented from multicasting messages while a view is being installed [FvR95].

2.3 Advantages of group communication

Group communication eases the development of distributed fault tolerant applications by encapsulating solutions to complex problems in simple abstractions. In fact, it can be observed that by using group communication, the intuitive solution to replication problems is also the correct solution. This is best captured by presenting examples of distributed fault tolerant applications and overview their solutions and correctness arguments.

2.3.1 Information dissemination

An interesting application of group communication is the problem of fault tolerant state dissemination. A server process keeps a set of data items that change frequently. Such changes have to be propagated to a set of observer processes that are interested in current values of items. The goal is to ensure that:

1. If the server ceases to modify its state, all observers eventually have the same values for all the items as the server.

2. If the server fails, all observers eventually have the same values for all the items and that the combined state is the state of the server at some previous instant.

The pattern described directly maps to information dissemination applications [PS97].

A solution using group communication is easily achieved. Each time the server changes the value of an item, it multicasts the identification of the item together with the new value to all observers. Upon delivery of a message, each observer updates the referred item with the new value. Figure 2.1 presents a sample run with a server S and two observers O_1 and O_2 updating the value of item x to x_1 and then to x_2.

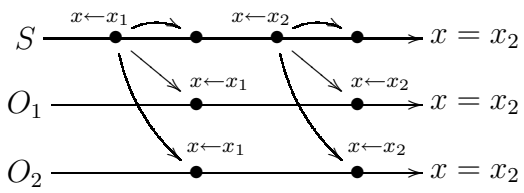


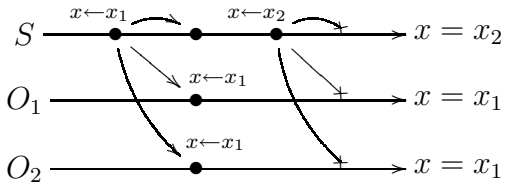
Figure 2.1: State dissemination using multicast.

Intuitively, the usefulness of reliable multicast can be illustrated by situations in which violation of any property of the specification leads to inconsistency. Such situations would have to be handled by the application when using unreliable multicast. Figure 2.2(a) depicts a run that violates Validity and thus fails to satisfy the first condition of the specification. Figure 2.2(b) shows that violating Agreement results in inconsistency regardless of the failure of the server. Violation of Integrity in Figure 2.2(c) allows duplicate delivery of messages and thus arbitrary return to any previous item value. Finally, Figure 2.3 illustrates the inconsistency resulting from delivering messages out of the context enforced by Reliable FIFO.

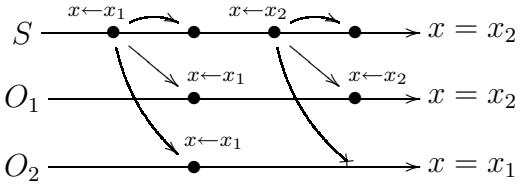
The correction of such implementation can be outlined as follows. Integrity and Reliable FIFO ensure that the sequence of messages delivered by any process is a prefix of the sequence of messages multicast by the server. Agreement ensures that all correct processes deliver the same sequence of messages. This is enough to satisfy second condition. Validity ensures that when the server is correct, it delivers exactly the same sequence of messages that was multicast, which is enough to satisfy the first condition.

2.3.2 Primary-backup replication

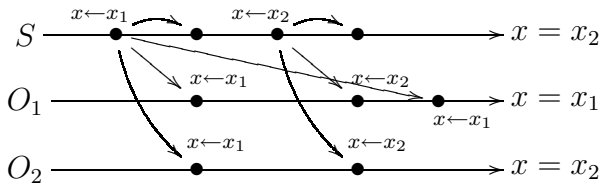
A common strategy for implementing strong consistent replication is the *primary-backup* approach to replication [BMST93, GS97a]: A single server – the



(a) Validity violation



(b) Agreement violation



(c) Integrity violation

Figure 2.2: Inconsistency resulting from unreliable multicast.

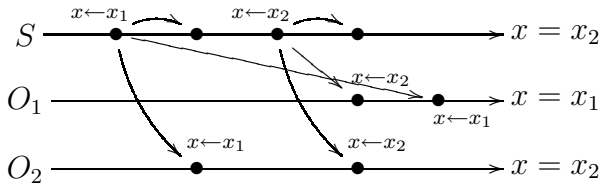


Figure 2.3: Inconsistency resulting from violation of Reliable FIFO.

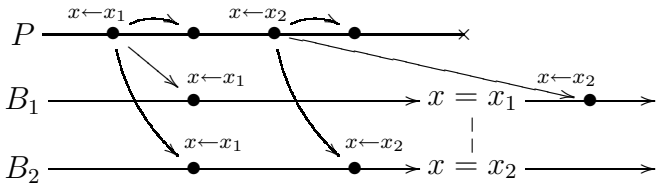


Figure 2.4: Inconsistency upon membership change without View Synchrony.

primary – handles requests from clients. Upon executing each request the primary broadcasts a state update to backup replicas. A reply can be sent to client after acknowledgment messages are collected from backup replicas. Should the primary fail, a backup replica takes-over as the primary.

Implementing primary-backup replication involves determining the primary and updating backup replicas. The primary can easily be selected using the group membership service, for instance, by deterministically selecting one member of the group, that is ensured to be the same for all participants.

Updating backups is an information dissemination problem and can be solved using the proposal of the preceding section. However, when the primary fails and a backup needs to take over, is must be determined whether a consistent state has already been reached, as it is possible that late messages are still in transit. Figure 2.4 illustrates a situation where despite enforcing reliable multicast and group membership, the state of processes upon view change is inconsistent. By using view synchrony the application can use membership notifications as indications that exactly the same messages have been delivered and thus that processes installing the new view are consistent.

2.3.3 Replicated state machine

An alternative strategy for replicating services is known as the *replicated state machine* [Sch93, GS97a]. In contrast to primary-backup replication, all repli-

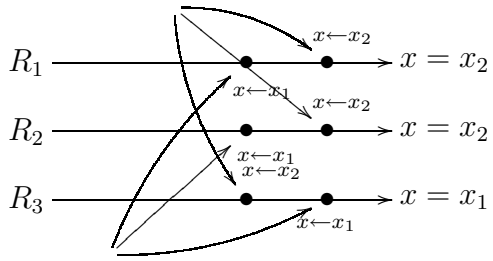


Figure 2.5: Inconsistency upon violation of Uniform Total Order.

cas handle requests directly from clients, executing them in parallel, updating their state and replying to clients. A client uses any of the replies.[2]

Consistency is ensured by the determinism of replicas and by processing the same sequence of client requests. The usefulness of reliable multicast as provided by group communication is again illustrated by Figure 2.2. Violation of Validity means that the client might not get any reply. Violation of Agreement and Integrity might leave servers with inconsistent state that results later in inconsistent replies to a request. In addition, implementing a replicated state machine using group communication requires also that requests are totally ordered. Otherwise, as shown in Figure 2.5, concurrent requests by multiple clients might be delivered by different orders.

Notice that ensuring that the same sequence of messages is delivered to correct processes is not enough. If a process delivers a different sequence, replies to a client, and then crashes, it exposes inconsistent state to a client. This is solved by requiring the multicast protocol to implement Uniform Agreement.

A specialization of the replicated state machine approach is the *database state machine* [Ped99, PGS02], that uses a replicated state machine to certify optimistically executed transactions. Execution of requests can therefore be done

[2]Assuming no byzantine faults.

in any of the replicas without locking. Read sets and write sets are then atomically multicast to all replicas that use a deterministic certification procedure to determine which requests can be committed and those that should be aborted due to conflicts with other transactions.

This makes better use of resources, as any replica can execute a transaction. Only certification, that is relatively inexpensive, has to be duplicated in all of them. To be deterministic, the replicated state machine approach alone would force a sequential execution of transactions, that is particularly inefficient. Primary-backup replication is also not adequate, as it would force all processing to be done by the primary. Neither would take advantage of the fact that transactions can be aborted and restarted later.

2.4 Challenges to group communication

Group communication is a convenient tool for programming fault tolerant distributed applications due to the strong guarantees provided. However, the same strong guarantees make it difficult to obtain good performance, in particular, in large and heterogeneous groups subjected to high throughput.

2.4.1 Scalability

A major issue in the scalability of reliable multicast protocols, and thus of group communication protocols, is the mechanism used to recover from lost packets. This requires a feedback mechanism that informs the sender of processes that have received each message. Gossip based protocols [BHO+99, EGH+01, EG02] and forward error correction (FEC) [NBT97] do not rely on feedback from receivers thus offering only probabilistic reliability guarantees.

In sender based protocols, the sender actively retransmits each packet to each destination until an acknowledgment message is received back. This is a simple and efficient mechanism for small groups, as immediate acknowl-

edgment results in low latency and optimal utilization of network resources. However, in large groups and with high throughput the overhead of transmitting and processing acknowledgment messages is not negligible. In fact, it can be a serious limitation to the performance and scalability of the protocol [ES98].

The alternative is a receiver based protocol, in which the sender does not perform retransmission unless this is explicitly requested by a receiver. The receiver must detect that a message has been lost and explicitly request its retransmission. This avoids the need for an explicit acknowledgment mechanism. In large groups, if a message is lost this also results in the sender being swamped in retransmission requests thus limiting performance [PTK94]. This can be avoided by allowing messages to be retrieved from processes other than the sender [BHO+99] and by performing retransmission to all receivers upon a single retransmission request, standing on the assumption that a message is likely to be lost by all or none of the receivers [FJL+97].

In addition, receiver based protocols need a mechanism to detect which messages have been received by all processes for garbage collection of buffers. This differs from positive acknowledgment as the sender is not required to identify which processes have not received a message: It just knows whether a message has been received by all. This allows the usage of mechanisms that are more efficient, requiring less network and processing resources to withstand high throughput. Examples of such mechanisms include the use of a logical token passing ring [AMMS+95] or gossiping [GHvR+97]. A comprehensive survey and comparison of such mechanisms is found in [Guo98]. The combined usage of a receiver based protocol with an efficient stability detection mechanism allows a reliable protocol to scale to large groups.

2.4.2 Throughput stability

The issue of achieving high and stable throughput in reliable multicast protocols has been addressed by recent research efforts [PS97, Bir99, BHO+99]: Perturbing the performance of a single participant in a multicast group degrades the performance of the whole group. This is a problem specially in large and heterogeneous groups where the probability of at any given time at least a single element being perturbed is high.

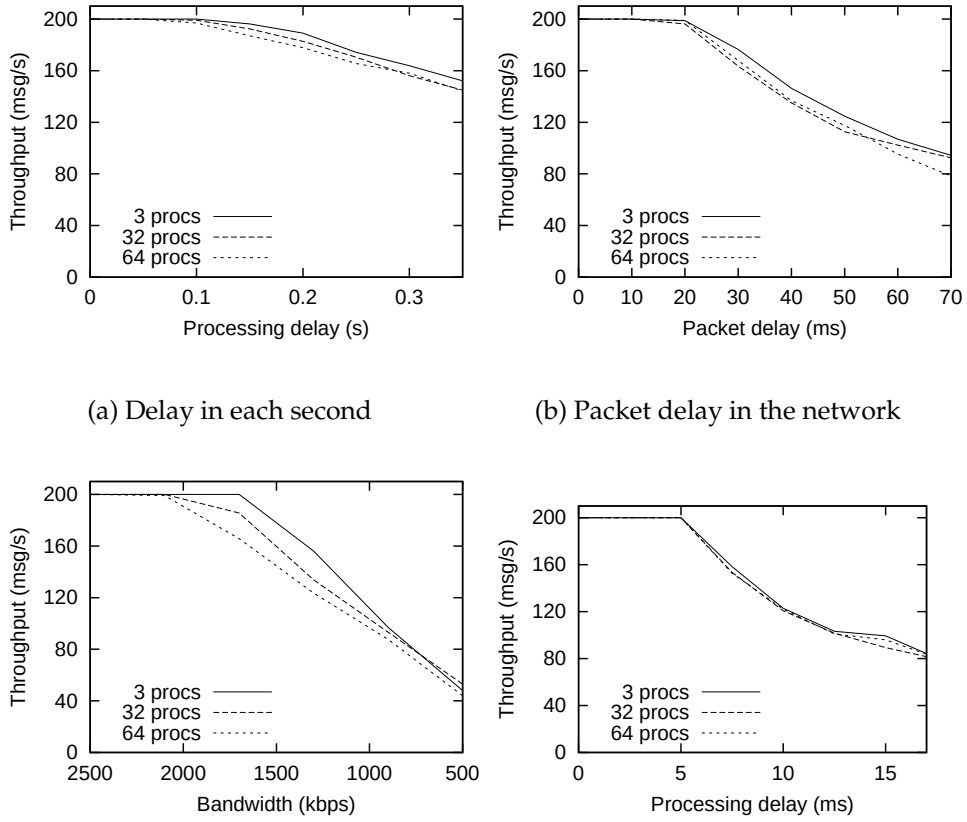
Throughput degradation is tightly related to strong reliability and flow control, and cannot be overcome by improving protocol mechanisms. To ensure strong reliability, each message might have to be retransmitted several times, possibly by different processes. Therefore, messages have to be buffered until their reception is acknowledged by all receivers, being then declared *stable* and subsequently garbage collected. If a single process is perturbed, it might delay the reception of a message. Even if the message is promptly received, perturbation might delay the acknowledgment of the fact. In both situations, messages have to be stored longer. If available buffer space is exhausted, the sender is blocked until enough buffer space is available to accommodate further messages. By periodically blocking the sender its throughput is reduced thus affecting all receivers.

To illustrate this we observe the behavior of a reliable multicast protocol when one element of the group is perturbed. Another element of the group is used as the sender producing messages at a constant rate of 200 msg/s. Then we measure and plot the throughput received by one of the remaining processes in messages per second (y axis) for different amounts of perturbation introduced (x axis).[3]

The first performance perturbation we consider is to make the protocol task sleep for an increasingly larger amount of time in every second.[4] The ex-

[3]Since at this point we are merely trying to motivate our work, we postpone a detailed description of the experimental setting to Chapter 5.

[4]This reproduces the results of [BHO+99].



(a) Delay in each second

(b) Packet delay in the network

(c) Network link bandwidth

(d) Processing delay upon delivery

Figure 2.6: Throughput degradation when a single receiver is perturbed.

periment is repeated with several group sizes, although always with only a single perturbed process. Figure 2.6(a) shows that although the reliable multicast protocol is scalable to large groups, a single perturbed receiver degrades the throughput of all the others. In addition, the same perturbation results in worse degradation in larger groups. Figure 2.6(b) shows that the impact of scalability is even larger when there is a delay on network packets to and from a single process. Figure 2.6(c) shows similar results obtained when the bandwidth of the link to the perturbed process is reduced.

Finally, we make the application task sleep between message deliveries.

In contrast to the previous results, this allows the protocol task to continue undisturbed exchanging control messages. Figure 2.6(d) shows that when the delay is too large to keep up with the sender (*i.e.*, it is greater than 5 ms), the sender is forced to wait, thereby decreasing its throughput and affecting all receivers. In contrast to previous results, different group sizes do not affect throughput.

Naturally, if reliability is strictly required, *i.e.*, if all recipients must eventually deliver all messages, either the sender adjusts to the slowest component or an indefinitely large buffer would be required. Although it is possible to exclude slow members from the group, transient problems by different machines may induce the same behavior. In such a situation, excluding members leads to reduced availability and additional performance impact when reintegrating them.

2.5 Relaxed reliability

As limitations of group communication are related to the enforcement of strong reliability guarantees, performance can often be improved by relaxing the guarantees provided. This is useful as it is actually possible to ensure consistency with relaxed guarantees. In this section we describe several approaches to relax reliability, order and view synchrony and their impact on the complexity of application programming.

2.5.1 Addressing throughput stability

Existing proposals addressing throughput stability weaken reliability guarantees, such that slower receivers are not required to deliver all messages and thus do not need to slow down the sender. Two major examples of this approach can be found in literature [BHO+99, CT90].

Bimodal multicast [BHO+99] offers probabilistic reliability: a message is

delivered to all or none of the processes with high probability. The probability of a message being received by only some of the processes can be made as small as necessary by adjusting configuration parameters. The underlying gossip-based protocol works by relaying messages a bounded number of times to randomly selected subsets of processes. After that, the message can be discarded from buffers without feedback from receivers, thus decoupling global throughput from the effect of perturbed processes. This means that messages might not be delivered to those processes that fail to meet performance assumptions. Notice that the protocol ensures that the application is notified when a message is omitted. The usefulness of the protocol is illustrated in the context of information dissemination applications.

The approach to protocol design known as Application Level Framing (ALF) [CT90] proposes that automatic buffering and retransmission should not be ensured by the protocol, therefore avoiding buffer starvation. Instead, the application is expected to explicitly request retransmissions of lost messages that are considered relevant and the sender is expected to provide them upon request of the protocol. The usefulness of such protocols has been described for file-transfer, where lost messages can be always reread from the immutable file, and for conferencing applications with loosely defined consistency requirements.

2.5.2 Consistency with relaxed reliability

When there are consistency requirements the additional complexity has to be managed by the application. A simple workaround for message loss that may compromise consistency is to force the receiver to exclude itself from the group and rejoin later in order to get a correct copy of the state [BHO+99]. This has however the same drawbacks as directly excluding slower members.

Excluding members can be avoided by retransmitting lost messages or other substituting messages as determined by the application. Consider the

information dissemination scenario of Section 2.3.1. The loss of an update message can be overcome by the delivery of a more recent message recreated from the current state of the sender. However, until this message is delivered, the observer has an incomplete sequence of messages that, upon failure of the sender, could result in inconsistency. Each observer has thus to buffer messages until a complete sequence is locally available and only then apply them.

The advantage of performing buffering within the application is that semantics can be used to minimize the amount of retransmission and buffering required. For instance, by recognizing that a lost message contained an obsolete value for an item, its retransmission can be avoided. Even if the obsolete message is already available, it can be immediately discarded, reducing the amount of buffer space used. The drawback is that this is not straightforward, even in the simple information dissemination application: Care must be taken that at all times a consistent state can be reached if the server fails.

Furthermore, when an observer is notified that a message has been lost, it might be unable to decide whether retransmission is required since it has no knowledge of that message's content and thus cannot evaluate whether the unknown message is relevant. This can be circumvented by the use of two multicast protocols in parallel [RM97]: An unreliable protocol used for payload and a reliable protocol used to convey meta-data describing the content of data messages sent on the payload channel. Using information from the control channel, the receiver may evaluate the relevance of lost messages in the payload channel and explicitly request retransmission when needed.

In short, relaxed reliability models proposed can in fact improve throughput stability but at the cost of more complexity for the application programmer. Namely, compared to the solution using group communication, the programmer is now forced to deal directly with process and network faults and manage buffers itself while managing a separate meta-data session.

2.5.3 Relaxed ordering and view synchrony

It is also possible to relax the guarantees of group communication in aspects other than reliability. Namely, several relaxed ordering and view synchrony criteria have been proposed.

A relaxation of order known as Generic Broadcast [PS99] stems from the observation that it is possible to ensure consistency in a replicated state machine despite some messages being delivered in different orders. For instance, messages that address disjoint parts of the state and do not interact. Relaxed ordering can be obtained by parameterizing the protocol with a *conflict relation*: Only pairs of messages that are related need to be totally ordered by the protocol. When the conflict relation is empty the protocol defaults to unordered reliable multicast and when all messages conflict it ensures total order. This allows the latency of the protocol to be reduced when conflicting messages are not multicast concurrently.

Ensuring that messages are received in the same view they were sent and that messages from correct processes are not discarded requires that processes are blocked while membership is changing [FvR95]. Although both are desirable characteristics, this results in exposing the latency of installing a new view to the application. This is often not acceptable and many protocols do not implement Sending View Delivery [ADKM92, Hay98]. An alternative is to inform the protocol of which messages need to be delivered in the view they were multicast and discard other messages when delivering them in the same view is not possible [SKM00].

2.6 Summary

This chapter motivates the work on semantic reliability by illustrating how strong reliability guarantees are responsible both for greatly simplifying reliable distributed system programming and for reduced performance of pro-

tocol implementations. Although existing proposals to alleviate performance problems result in substantially higher complexity to be managed by applications, they show that application semantics are effective in improving performance.

It is interesting to note that relaxations of ordering and view synchrony do not dramatically change group communication. In fact, unless the application program specifies in which situations the guarantees can be relaxed, both proposals default to strong guarantees. This means that the application can be easily developed using strong guarantees and later, incrementally, have its performance improved by giving additional details to the protocol about the messages being exchanged.

This contrasts to reliability relaxations that from the start require that the application is concerned with complex mechanisms for buffering and retransmission on which its correctness depends. The challenge is thus to improve the performance using application semantics to relax reliability without discarding the comfortable programming paradigm of group communication.

Chapter 3

Message Obsolescence

We propose to selectively relax reliability using knowledge about message semantics, specifically, by determining which messages become obsolete while still in transit. This improves throughput stability without burdening the application with all the complexity of maintaining consistency that is supposed to be encapsulated in the group communication protocol. To use this concept we must capture message semantics at the application level and convey it to the protocol. In this chapter we discuss how message semantics can be formalized, which data-structure is appropriate to represent it, and how this structure is built in common application scenarios.

3.1 Selectively relaxing reliability

Allowing applications to determine which messages to omit in slower receivers improves throughput stability, but doing this in a fault tolerant manner with existing relaxed reliability models highly complicates application programming. In fact, allowing the application to dictate the policy for discarding messages is done by burdening the application with mechanisms for buffering and retransmission. If enough knowledge about message semantics can be conveyed to the protocol, it becomes possible to select which messages can

be omitted by slower processes without endangering consistency and without bringing the complexity of buffering and retransmission into the application.

3.1.1 Intuition

The basic idea behind our approach is that in a distributed application some messages become obsolete while still in transit because other messages overwrite or implicitly convey their content. If a message that has not yet been delivered to the application is recognized as obsolete, it can be safely purged from buffers without compromising the application's correctness. Notice that when the system is congested this is likely to happen, as buffers in the path to the slower component will be full and thus contain messages sent some time ago. Over time, if enough obsolete messages can be purged, the sender never needs to be blocked thus sustaining a high throughput. If not, purging some messages at least ensures that blocking is necessary less often and for shorter periods of time.

Processes that have not delivered messages that became obsolete and are purged omit their delivery. The result is a reliability criterion in which the protocol ensures that message content is delivered either explicitly or implicitly in more recent messages. This means that the protocol reliably transmits semantics of message sequences despite the loss of individual messages, hence *semantic reliability*.

3.1.2 Applications

Applications embodying operations with overwrite semantics, in particular, applications managing read-write items are the most obvious example of applications that exhibit message obsolescence. In these applications, any update of a given item is made obsolete by subsequent update operations. Recognizing this fact, some applications deal with obsolescence directly. For instance, distributed file-systems, such as NFS, cache write operations in the

client to minimize network traffic [Sat90]. Another example are weakly consistent distributed shared memory systems, where the effect of memory operations is restricted by synchronization primitives. In practice, this allows delaying and even suppressing the dissemination of updates [RM93].

Although this solves the problem when using point-to-point communication channels, it is not possible to implement these optimizations at the application level when using multicast protocols. In order to timely convey the message to faster receivers, the application forwards updates to the multicast protocol as soon as possible. At that point, the message becomes out of reach of the application and cannot be discarded even if shortly after it becomes obsolete and thus unnecessary load to already slower receivers.

Typical examples are applications such as on-line trading systems, where new quotes have to be continuously disseminated to a large number of recipients [PS97] or distributed multi-player games in which frequently updated game state is disseminated to a group of replicas (see Chapter 6). This is also the case of control and monitoring applications in which the input from sensors is frequently updated [Bir99].

Not only applications with read-write semantics exhibit the obsolescence property. For instance, many distributed algorithms are structured in logical rounds and, when the algorithm advances to the next round messages from previous rounds become obsolete. Recognizing this property, the concept of *stubborn channel* has been proposed [GOS98], in which reliability has to be ensured just for the last messages (note that the number of rounds is not bounded). It has been shown that the fundamental problem of distributed consensus [GOS98, Oli00] can be solved in asynchronous distributed systems augmented with failure detectors and $1-$*stubborn channels*. Stubborn channels can be seen has a particular case of obsolescence.

3.2 Expressing message obsolescence

Semantic reliability requires that the protocol is able to determine which messages are obsolete based on their semantics. A convenient mechanism is to perform the notification of obsolescence of some message using another message whose content is directly responsible for the fact. For instance, the message carrying the updated value of an item implicitly notifies the protocol that the preceding message for the same item has become obsolete.

All required semantics can thus be abstracted as a binary relation on the set of messages. The fact that a message m is obsoleted by a message m' is denoted as $m \sqsubset m'$. The intuitive meaning of the *obsolescence relation* is that if $m \sqsubset m'$ and m' is delivered, the correctness of the application is not affected by omitting the delivery of m. The notation $m \sqsubseteq m'$ is used as a shorthand for $m \sqsubset m' \vee m = m'$.

Specification, implementation and usage of protocols based on message obsolescence is simplified by making the obsolescence relation behave according to intuition that messages are obsolete because they are implicitly conveyed or overwritten by other messages. Therefore, it is assumed that the obsolescence relation is transitive and that it is a subset of the causal order relation. The obsolescence relation is thus a *strict partial order* relation.

A binary relation on the set of messages is not the only option to capture message semantic. For instance, an obsolescence relation $\sqsubset^*$ among sets of messages could be considered, allowing direct representation of facts such as "m becomes obsolete upon delivery of both m' and m''" with the notation $\{m\} \sqsubset^* \{m', m''\}$. The additional complexity is however not worthwhile, as the same result can be achieved by using an additional ordering relation, that is already true for most applications based on group communication. Consider for instance $\{m_1, m_2\} \sqsubset^* \{m_3, m_4\}$, where delivery of both m_3, m_4 is required to make m_1, m_2 obsolete. By relying on delivery order to ensure that the delivery sequence is m_3, m_4, this reduces to $m_1 \sqsubset m_4$ and $m_2 \sqsubset m_4$, as the

delivery of m_4 implies previous delivery of m_3. Further examples are given in Section 3.4.

3.3 Representing message obsolescence

Implementing semantic reliability requires that an interface is exposed for the application to convey the obsolescence relation to the protocol. We are interested in general purpose techniques that can be applied to a wide range of systems in an efficient manner. Therefore we exclude solutions such as requiring special formatting of messages [CRW00] for the protocol to infer which messages are obsolete.

The obsolescence relation has also to be available in processes other than the sender of the messages and possibly in processes not running the application but acting only as routers. Therefore we exclude also callbacks to query the application for obsolescence. Time and space efficient algorithms and data structures to manipulate protocol buffers and determine obsolete messages exclude also the possibility of enriching the messages with code [TTP+95]. The remaining option is to use a concrete representation of the obsolescence relation as a data structure that is used to annotate messages. This data structure is initialized by the application and supplied to the protocol along with each message in the multicast operation.

A requirement of the representation technique is that it is able to represent a large subset of possible obsolescence relations to maximize the ability to recognize and purge obsolete messages. On the other hand, the representation does not need to be complete to ensure application correctness: if some messages are not recognized as obsolete, only purging efficiency is lost. The amount of relations that can be represented has therefore to be weighted against the need to make the structure compact and efficient, as annotations need to be stored in buffers, transmitted over the network and manipulated

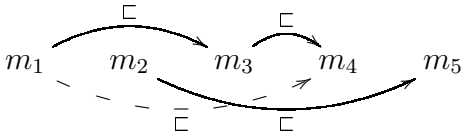


Figure 3.1: Sample obsolescence relation.

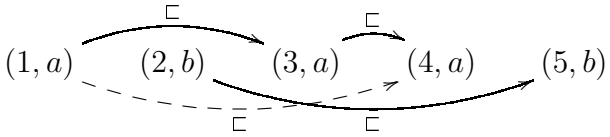


Figure 3.2: Representation of Figure 3.1 with item tagging.

when searching for obsolete messages.

3.3.1 Item tagging

The simplest representation technique for applications managing read-write data items consists in associating a unique tag to each data item. Each message is tagged with the identifier of the data item it is updating. Tags are added to the message headers and used in combination with the sequence numbers generated by the protocol: if two messages from the same sender carry the same tag, the one with the highest sequence number makes the other obsolete. This allows relations among messages originating from the same sender to be represented.

Let the header of a message m be represented as (*seqNum*,*itemId*) then $m \sqsubset m'$ if $m.\mathit{seqNum} < m'.\mathit{seqNum}$ and $m.\mathit{itemId} = m'.\mathit{itemId}$. The representation of a sample obsolescence relation of Figure 3.1 is presented in Figure 3.2 assuming that m_1, m_3, m_4 update item a and m_2, m_5 update item b. Notice that although simple, this technique does not allow expressing that a message

obsoletes several other non-related messages. For instance, it would not be possible to represent a further message m_6 such that simultaneously $m_4 \sqsubset m_6$ and $m_5 \sqsubset m_6$, as $m_4 \not\sqsubset m5$.

Notice that *seqNum* can be determined by the protocol itself and thus the protocol interface needs only a single additional parameter for the *itemId*. This technique can be generalized to represent obsolescence among causally related messages originating from multiple senders: one has to use a vector instead of a single sequence number. As happens with the sequence number, the vector can be the same used for causal ordering thus not incurring in any additional overhead.

3.3.2 Message enumeration

A more general alternative consists in having each message explicitly enumerate which preceding messages it makes obsolete. This approach is clearly more expressive than the item tagging approach and can easily be used to represent relations between causally related messages from distinct senders. On the other hand, this results in significant space overhead in message headers to list message identifiers.

In addition it burdens either the protocol or the application program with the task of determining the transitive closure of the relation. Consider three messages such that $m_1 \sqsubset m_2 \sqsubset m_3$. The coding of obsolescence should allow to verify that $m_1 \sqsubset m_3$ without requiring m_2 to be available. For instance, because m_2 has already been purged. To ensure that the transitivity of the obsolescence is preserved in the message enumeration technique, a message must enumerate not only its direct predecessors, but all the (transitive) predecessors.

If the header of a message m is represented as (*id*,*predSet*) then $m \sqsubset m'$ if $m.id \in m'.predSet$. Considering only messages from the same sender, the identification is a single integer. The representation of the sample obsoles-

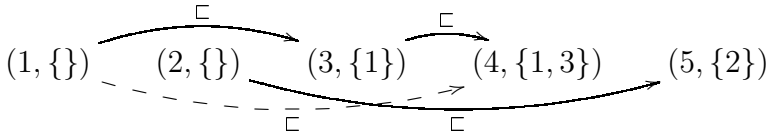


Figure 3.3: Representation of Figure 3.1 with message enumeration.

cence relation of Figure 3.1 is presented in Figure 3.3. This technique trivially expresses that a message obsoletes several other non-related messages. For instance, it would be possible to represent a further message m_6 such that simultaneously $m_4 \sqsubset m_6$ and $m_5 \sqsubset m_6$ with $(6, \{1, 2, 3, 4, 5\})$.

Notice that *id* can be assigned by the protocol itself and be opaque to the application, as long as the application is informed upon multicast and delivery of each message. The protocol interface has also to be extended to accept *predSet* for each message multicast.

In practice, only those messages that are possibly still in transit need to be enumerated. Furthermore, only recent messages from the enumeration need to be carried by each message without any significant impact on the purging efficiency. This optimization is possible because it is very unlikely that two messages far apart in the message stream can be found simultaneously in the same buffer.

3.3.3 k-Enumeration

The k-enumeration technique is a representation strategy that combines the efficiency and simplicity of the tagging approach with the expressiveness of the message enumeration approach. The technique exploits the facts that purging is mainly applied to pairs of messages that are close to each other in the message stream and that messages related by obsolescence are totally ordered in a sequence. If the supporting order is FIFO, it can be used only to represent ob-

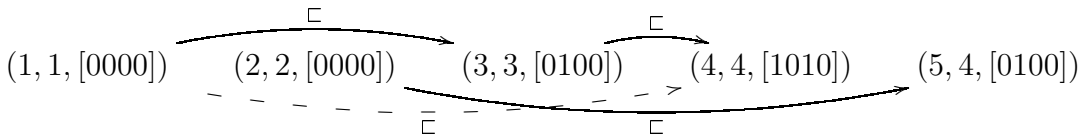


Figure 3.4: Representation of Figure 3.1 with k-enumeration.

solescence among messages from the same sender. If the supporting order is a total order, it can be used to represent obsolescence among causally related messages from multiple senders.

Each message explicitly enumerates which of the k preceding messages in the supporting order it makes obsolete. This information can be stored in a bitmap of k bits. If the n^{th} position of the bitmap is set to 1, the message makes obsolete the n^{th} preceding message. This results in a compact representation that is also efficiently manipulated: it is very easy to compute the representation of transitive obsolescence relations using only shift and binary "or" operators. It also makes it very easy to compute, using the same efficient operators, the representation of the obsolescence relation when a message makes several other obsolete.

Let the header of a message m in a sequence be (*seqNum*,*base*,*map*). Bits in *map* are indexed from 1 to k. Then $m \sqsubset m'$ if $m'.map[m'.base - m.seqNum] = 1$. Notice that for the obsolescence relation to be coherent with causality, it is always true that $base \leq seqNum$. The representation of the sample obsolescence relation of Figure 3.1 with $k = 4$ is presented in Figure 3.4. The bitmap is ordered from left to right and thus $[1000]$ refers to the message with *seqNum* equal to $base - 1$. This technique trivially expresses that a message obsoletes several other non-related messages. For instance, it would be possible to represent a further message m_6 such that simultaneously $m_4 \sqsubset m_6$ and $m_5 \sqsubset m_6$ with $(6, 6, [1111])$. Notice that $m_1 \sqsubset m_6$ cannot be determined and thus must be assumed as not true by the protocol.

Both *seqNum* and *base* can be determined by the protocol itself. If the supporting order is FIFO, they are equal to the position of the message in the sequence and thus only one needs to be stored. If the supporting order is total order, *base* is the sequence of the last message delivered and *seqNum* is attributed by the total order protocol.

3.4 Programming examples

Any application of reliable multicast can be used with semantic reliability by using an empty obsolescence relation. This does not however allow any improvement in throughput stability as no message ever becomes obsolete and thus is ever purged. To take advantage of a semantically reliable protocol a suitable obsolescence relation has to be determined and then conveyed to the protocol.

In this section we give examples of these two steps for some common application constructs: state dissemination and a replicated state machine. The technique used to represent the obsolescence relation is k-enumeration. An informal definition of the protocol is presented just to convey the intuition and allow reasoning on the correctness of the application with a non-empty obsolescence relation.

3.4.1 Informal protocol definition

An informal definition of a semantically reliable multicast protocol for finite sequences of messages can be obtained by using a reliable multicast protocol for comparison:

Semantically Reliable Multicast: Consider the sequence S_p of messages delivered by each correct process p using semantically reliable multicast. There is a sequence of messages R that in the same situation (*i.e.*, the same messages

R	Mandatory in S_p	Optional in S_p
m_1	m_1	
m_1, m_2	m_1, m_2	
m_1, m_2, m_3	m_2, m_3	m_1
m_1, m_2, m_3, m_4	m_2, m_4	m_1, m_3
m_1, m_2, m_3, m_4, m_5	m_4, m_5	m_1, m_2, m_3

Figure 3.5: Possible runs with the obsolescence relation of Figure 3.1.

multicast and the same correct processes) would have to be delivered to all correct processes by reliable multicast. Sequences R and S_p must satisfy the following properties:

P1. Every message m in some S_p is also in R;

P2. For every message m in R and for every correct process p there is some message m' in S_p such that $m \sqsubseteq m'$;

P3. If two messages m, m' appear both in R and in some S_p then they do so in the same order.

This definition does not specify which ordering is enforced by the reliable multicast protocol. Therefore, depending on the ordering chosen for reliable multicast used, a different semantically reliable counterpart exists. For instance, by assuming that R is a sequence, this definition assumes a total order on messages: Either because there is a single sender and the protocol enforces FIFO or because the protocol enforces total order. We use both FIFO and total ordered versions of semantically reliable multicast in the examples.

Consider the sample obsolescence relation of Figure 3.1. If all processes are correct, then R is the sequence m_1, m_2, m_3, m_4, m_5. This means that for every correct p, S_p includes m_4 and m_5. Including m_1, m_2, m_3 is optional, as long as

```
    /* Server process */
    Initially:
        seq = 0
        for all id: info[id].map = [0..0] and info[id].seq = 0

1:  upon update(id, value) do
2:      write(id, value)
3:      seq ← seq+1
4:      info[id].map ← SHIFT(info[id].map, seq − info[id].seq)
5:      multicast(UPDATE(id,value)) tagged with (seq, seq, info[id].map)
6:      info[id].seq← seq+1
7:      info[id].map← SHIFT(info[id].map, 1) OR [10..0]

    /* Observer processes */
8:  upon deliver(UPDATE(id, value)) do
9:      write(id, value)
```

Figure 3.6: Pseudo-code of state dissemination.

the relative order in the sequence is preserved. If the sender fails, then any prefix is acceptable as R and S_p varies accordingly as shown in Figure 3.5.

This definition avoids the complexity of a precise specification and is useful only for motivation. Notice that the consequences of semantic reliability on liveness are not addressed, *e.g.*, which messages should be delivered if an infinite sequence of related messages is multicast. Nevertheless this suffices for reasoning about the safety of programming examples presented below. In fact, as with strong reliability, several different specifications are possible. Chapter 4 will present and discuss formal specifications and algorithms.

3.4.2 Single item operations

The solution proposed for fault tolerant state dissemination using FIFO reliable multicast can directly be reused with FIFO semantically reliable protocol by assuming that an update to some item makes all previous updates to the

same item obsolete. This leads to the following definition of the obsolescence relation: Let UPDATE(x, v) denote a message carrying an update of item x to value v. If $m_i = \text{UPDATE}(x, v)$ and $m_j = \text{UPDATE}(y, u)$ are the i^{th} message and the j^{th} message in a sequence, $m_i \sqsubset m_j$ if $i < j$ and $x = y$. The assumption that all operations are updates can easily be removed as described in Chapter 6.

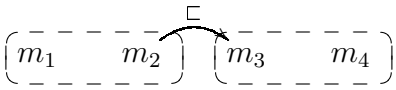
It is easily observed that the value of an item after delivering a message sequence S_p is the same as after delivering R: If the item has been modified, its value is that of the latest update. As the last update of each item never becomes obsolete, then it is in S_p (by P2 and P1) and it is the last update of the item in S_p (by P3 and P1). If the item is not modified by R, then it is also not modified by S_p (by P1) and thus its value is the initial value.

The next step is to represent the obsolescence relation to be conveyed to the protocol. Figure 3.6 presents pseudo-code for that using the k-enumeration technique. The additional lines required only with semantic reliability are shaded. Function *SHIFT*(b, n) used in lines 4 and 7 shifts right a bitmap b by n bits. *OR* and *AND* denote bitwise logical operations on the bitmap. $[0..0]$ denotes a bitmap with all bits cleared to 0. $[10..0]$ denotes a bitmap with only the leftmost bit set to 1 and $[01..1]$ a bitmap with only the leftmost bit set to 0.

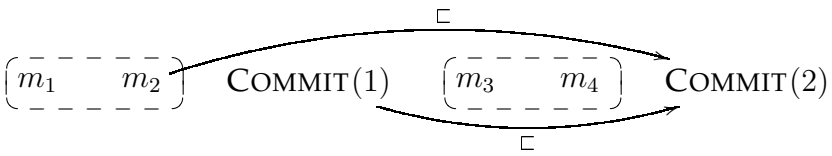
Each update operation has a sequence number calculated in line 3. For each item with identification id, the algorithm stores a bitmap *info[id].map* that holds a k-enumeration of previous operations that have modified its value. Variable *info[id].seq* holds the sequence number of the latest of these.

As *info[id].map* refers to the immediate predecessors of update with sequence *info[id].seq*, it is shifted to make it refer to the predecessors of current operation *seq* (line 4). The result is used directly as the tag to the multicast operation (line 5). In line 7, *info[id].map* is updated once more by inserting 1 in its leftmost bit, referring to the current update.

Notice that the additional operations required are all elementary arithmetic and logical operations that are quickly executed. The computational effort required to compute the representation of the relation is thus negligible



(a) Intended obsolescence relation



(b) Modification to preserve operation boundaries

Figure 3.7: Obsolescence relation preserving operation boundaries.

compared with the cost of multicasting a message.

If an item is not updated in k consecutive operations, the value of *info[id]* is meaningless as *info[id].map* after line 4 it will be entirely set to zero. It is therefore possible to store *info[id]* only for items modified by the latest k operations, thus saving space in exchange of some time to address *info[id]*.

Notice also that the following modification to the obsolescence definition would not make any difference in what messages can be purged: Make $m \sqsubseteq m'$ if there is some m'' such that m'' causally precedes m' and $m \sqsubset m''$. This would avoid separately storing a bitmap for every item and save space. We make use of this in the following section.

3.4.3 Multiple item operations

A slightly more complex variation of the state dissemination problem is when each operation atomically updates several items. This imposes an additional restriction on consistency that requires updates of the same operation to be applied atomically by all observers. Consider the example of Figure 3.7(a) where m_1, m_2 and m_3, m_4 are updates resulting from two operations. There-

fore applying just m_1 or m_1, m_2, m_3 will not result in a consistent state.

This can be solved with FIFO reliable multicast either if all updates from an operation are grouped in a single message or if individual updates can be buffered by the observer until the full operation is locally available and only then applied. Should the server fail, the state of each of the observers reflects the last fully delivered operation.

Neither of these can be used with semantic reliability with the technique shown in the previous section. Grouping several updates within a single message drastically reduces purging opportunities, as $m \sqsubset m'$ only if m updates a subset of the items updated by m'. On the other hand, delaying the application of individual updates may result in inconsistency. For instance in Figure 3.7(a) with $m_2 \sqsubset m_3$, upon failure of the sender it is possible that an observer delivers m_1, m_3, and thus does no apply any of them, and that simultaneously another observer delivers m_1, m_2, m_3 and thus applies m_1, m_2.

This can be avoided by modifying the obsolescence relation with the introduction of additional COMMIT(r) messages marking the boundary of each operation r. With this, it is possible to postpone making a message obsolete until the full operation is ensured to be delivered. Figure 3.7(b) shows the modified sample relation. Notice that all runs including m_3 but not m_4 will include m_2 and COMMIT(1).

The obsolescence relation is defined as follows. Considering that the i^{th} message in a sequence is an update of item x to value v as part of operation r is denoted $m_i = \text{UPDATE}(r, x, v)$, then $m_i \sqsubset m_j$ if either:

- $m_i = \text{COMMIT}(r)$, $m_j = \text{COMMIT}(s)$ and $r < s$; or
- $m_i = \text{UPDATE}(r, x, v)$, $m_j = \text{COMMIT}(s)$ and some $m_k = \text{UPDATE}(t, y, u)$ exists such that $x = y$ and $r < t \leq s$.

Notice that any commit message makes all previous commit messages obsolete, even if corresponding update messages refer to distict items. This hap-

```
    /* Server process */
    Initially:
        seq = 0 and tseq = 0
        map = [0..0] and for all id: iseq[id] = 0

1:  upon update(set) do
2:      write_all(set)
3:      tseq ← tseq+1
4:      for all (id,value) ∈ set do
5:          seq ← seq+1
6:          map ← SHIFT(map,1) OR SHIFT([10..0], seq − iseq[id])
7:          iseq[id] ← seq
8:          multicast(UPDATE(tseq,id,value)) tagged with (seq, seq, [0..0])
9:      seq ← seq+1
10:     map ← SHIFT(map, 1)
11:     multicast(COMMIT) tagged with (seq, seq, map)
12:     cseq ← seq+1
13:     map ← SHIFT(map,1) OR [10..0]

    /* Observer processes */
    Initially:
        queue is empty
        last = 1

14: upon deliver(UPDATE(tseq, id, value)) do
15:     if tseq ≠ last
16:         write_all(queue)
17:         clear queue
18:     queue(id, value)
19:     last ← tseq+1

20: upon deliver(COMMIT) do
21:     write_all(queue)
22:     clear queue
23:     last ← tseq+1
```

Figure 3.8: Pseudo-code of state dissemination preserving operation boundaries.

pens because all that is required is that eventually a commit message is delivered, resulting in all receivers eventually reaching a consistent state.

The representation of this obsolescence relation is done by the shaded lines in Figure 3.8. Upon an update request, all values are updated (line 2) and then multicast one by one to observers (line 8). Finally a commit marker is multicast (line 11). Observers deliver updates one by one and store them in a queue (lines 15 to 18). Upon commit, all updates are applied and the queue is emptied (lines 21 to 23). Calculating tags for semantic reliability involves three steps:

- while multicasting updates with an empty tag (line 8), the obsolescence map is updated with bits referring messages made obsolete by the current operation (lines 6 and 7);

- the commit message is tagged with *map* adjusted to current message sequence (line 11), which stores the messages made obsolete up to the previous commit operation;

- the bitmap of obsolete messages for operation $tseq+1$ is saved (line 13), including the current commit message.

Notice that multiple application messages can be clustered in a single transport level message for efficient network transmission and thus do not introduce additional overhead. This is a feature that is present in group communication toolkits as it is invaluable regardless of semantic reliability [HvR95].

Notice also that the role of commit messages can be performed by a single bit that is set in the last message of each operation. Nevertheless, this is not critical, as in a message sequence all but the last commit message are obsolete and can be purged.

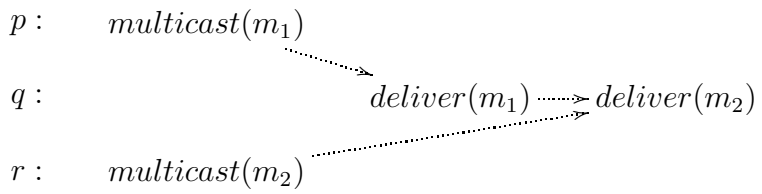
3.4.4 Concurrent operations

In this section we consider obsolescence relations among concurrent messages. As an example we use a set of servers managing a collection of items as a replicated state machine: Clients can request the current value of an item or update the item with a new value. With group communication, this can be solved as described in Section 2.3.3.

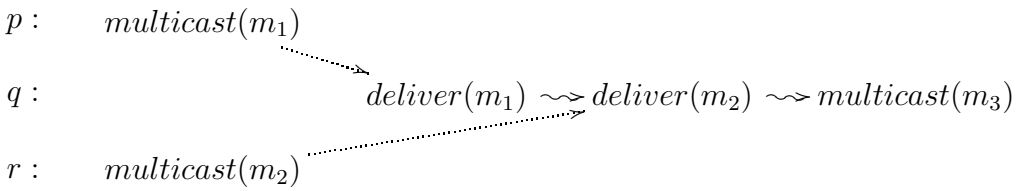
Obtaining performance improvements from semantic reliability requires that a non-empty obsolescence relation is defined. It is thus desirable that update messages can be made obsolete by later update requests. In contrast to previous examples, using obsolescence relations among messages originating from the same sender is not interesting. In fact, it may lead to inconsistency. Consider messages m_1, m_2 from client process p_1 updating the same item x. If $m_1 \sqsubset m_2$, it is possible that some servers deliver both m_1, m_2 and others only m_2. However, it is possible that there is some other client process that issues a message m_3 requesting the value of x. If the total order is m_1, m_3, m_2 then some servers will reply with the initial value of x while other reply with the value established by m_1.

Furthermore, a pair of messages m_1, m_2 addressing the same item might have been issued by different processes and thus concurrently as shown in Figure 3.9(a). Although messages m_1 and m_2 update the same item no relation among them can be established because this relation cannot be known at the time m_2 is multicast. That $m_1 \sqsubset m_2$ can only be established after the protocol decides to order m_1 before m_2.

Both these problems can be circumvented by relying on a third message that is multicast by some process after delivering both m_1 and m_2. For instance, Figure 3.9(b) shows that process q upon multicasting m_3, which by ordering will delivered after m_2, may notify the protocol that m_1 has become obsolete. This will allow the same messages to be purged as the originally intended relation.



(a) $m_1 \sqsubset m_2$ cannot be represented because $multicast(m_1)||multicast(m_2)$



(b) $m_1 \sqsubset m_3$ can be represented and has the same effect ($\rightsquigarrow$ represents useful causality)

Figure 3.9: Obsolescence relations among concurrent messages.

The process that multicasts m_3 already knows if between m_1 and m_2 there is a message whose external effects depends on the value of item x. Should there be some message that reads the value of x, no process makes m_1 obsolete upon multicasting m_3.

Considering that $\rightsquigarrow$ denotes causality, that $m_i = \text{UPDATE}(x, v)$ denotes the i^{th} message carrying an update of item x to value v, and $m_l = \text{GET}(z)$ the l^{th} message carrying a request for the value of item z, then $m_i \sqsubset m_j$ if there is some message $m_k = \text{UPDATE}(y, u)$ such that both:

- $deliver(m_i) \rightsquigarrow deliver(m_k) \rightsquigarrow multicast(m_j)$ and $x = y$; and
- there is no message $m_l = \text{GET}(z)$ such that $x = z$ and $deliver(m_i) \rightsquigarrow deliver(m_l)$ and $deliver(m_l) \rightsquigarrow deliver(m_k)$;

Figure 3.10 presents the pseudo-code of server processes. Client processes

```
    /* Server processes */
    Initially:
        seq = 0 and map = [0..0]
        for all id: info[id].map = [0..0] and info[id].seq = 0

1:  upon deliver(UPDATE(id, value)) do
2:      write(id, value)
3:      seq ← seq+1
4:      info[id].map ← SHIFT(info[id].map, seq − info[id].seq)
5:      map ← SHIFT(map,1) OR info[id].map
6:      info[id].seq ← seq+1
7:      info[id].map ← SHIFT(info[id].map, 1) OR [10..0]

8:  upon deliver(GET(id)) do
9:      info[id].map ← info[id].map AND [01..1]
10:     seq ← seq+1
11:     map ← SHIFT(map,1)
12:     multicast(REPLY(id,read(id))) tagged with (?,seq,map)

13: upon deliver(anything else) do
14:     map ← SHIFT(map,1)
15:     seq ← seq+1
```

Figure 3.10: Pseudo-code of a replicated state machine.

are not displayed. Upon receiving a request to update an item, a server updates its current map of obsolete messages *map* with previous updates to the same item stored in *info[id].map* (lines 4 and 5). It then updates *info[id]* to include the update message that is being delivered (lines 6 and 7).

When handling a request to read the value of an item, the last update is erased from *info[id]* (line 9) ensuring that it will never become obsolete. The reply is then issued (line 12) taking advantage of the opportunity to notify the protocol of messages known to be obsolete. Additional messages could be used specifically for the purpose of notifying the protocol of which messages became obsolete, instead of using reply messages. As each of these messages

would make the previous one obsolete, this would not represent significant overhead.

Notice also that, upon multicast of a message in line 12, the full annotation of the message is not known. It is the responsibility of the protocol to set *seqNum* to the sequence number of the message after it has been totally ordered.

In more complex applications, the distinction between update and read operation might not be clear. For instance, a single operation might be used to update the value of some item with the result of a computation with the values of other items. In such situation, it will be necessary to reset the maps of all items read as in line 9 while performing the update operation. Notice however that maps need only to be reset if the computation results in state modification or is externally visible.

This is better illustrated with a concrete example. Consider an application used in on-line trading to match seller and buyer bids. Replicated servers maintain the current portfolio of each client as well as current seller and buyer bids. Clients may at any time update their bids to buy or sell a specific stock. If a seller and buyer bids match, the transaction is performed by updating the portfolios and marking the bid items as read in order not to become obsolete. If not, bid items need not be marked as read and the corresponding message can later become obsolete.

Another example are control applications, where the input of multiple sensors is processed by a replicated controller. Sensor readings that do not trigger actuators can be made obsolete by subsequent readings. This is useful when sensors are monitoring a physical variable that changes continuously (*e.g.*, the position of an airplane) that may be critical but is seldom used to decide the answer to another request (*e.g.*, an airplane has changed sectors).

3.5 Summary

This chapter presents the intuition underlying semantic reliability: some messages can be discarded because they become obsolete while still in transit. All semantics required for the protocol to determine obsolescence can be captured as a binary relation on messages. This is the basis for the formal definition of the semantically reliable group communication in the following chapter.

This approach is motivated by discussing an appropriate programmer's interface for semantically reliable protocols, which allows the obsolescence relation to be conveyed by the application to the protocol. The usage of this interface and of semantic reliability in general is illustrated with concrete examples, namely, information dissemination and a simple replicated state machine.

Chapter 4

Semantically Reliable Protocols

In this chapter we present the specifications and algorithms for semantically reliable multicast protocols. Specifically, a multicast protocol requiring a correct sender (S-SRM) is used to introduce semantic reliability; the core of the protocol suite is the semantically reliable multicast (S-RM); and the interaction with group membership is illustrated with semantic view synchronous multicast (S-VSM). These protocols can be adapted to provide uniform agreement, causal order and uniform total order in a similar fashion to conventional group communication.

4.1 System model and notation

Protocol specifications are presented in the context of an asynchronous message passing system model augmented with a failure detector [CT96]. The distributed system is modeled as a set of sequential processes that can: send a message; receive a message; consult the failure detection oracle; perform a local computation; and crash. No assumptions are made on relative execution speed of processes or on the existence of synchronized clocks. Processes can only fail by crashing and do not recover, thus excluding byzantine faults. A process that does not crash is correct. We assume that at most a minority f of

processes can crash.

Processes are fully connected by an asynchronous network of point-to-point message passing channels. Asynchrony means that there is no bound on the time that a message takes to be transmitted. A channel connecting process p to process q is used through primitives *send*(m, q) in process p and *receive*(m, p) in process q. Briefly, reliability means that if both the sender and receiver processes are correct, the message is eventually received. Additionally we assume that channels are FIFO ordered. Assumptions on reliability and FIFO order are actually not strictly required, but used to simplify the presentation of the algorithms. In fact, channels can be reduced to fair-lossy channels [BCBT96].

A consensus protocol [CT96] is assumed to be available and modeled as a function *consensus*(v) that takes as parameter a proposed value and returns the decided value. Informally, consensus ensures that all correct processes eventually decide the same value and that the decided value is one of the proposed values. Notice that consensus can be solved in our model, *i.e.*, with unreliable failure detection [CHT96] and even if the assumption on reliability of channels was relaxed [GOS98].

The multicast service is used through a pair of primitives: *multicast*(m) and *deliver*(m). If during a computation a process executes *multicast*(m) (resp. *deliver*(m)) it is said to "multicast message m" (resp. "deliver message m"). When defining a view synchronous multicast, view changes are signaled to the application by the *install*(v) primitive. Each view notification v includes the identification of the view *id*(v) and of the set of processes which constitute the current membership of the group *memb*(v).

The obsolescence relation is used as described in Section 3.2. Briefly, the fact that a message m is obsoleted by a message m' is denoted as $m \sqsubset m'$. The notation $m \sqsubseteq m'$ is used as a shorthand for $m \sqsubset m' \vee m = m'$. The obsolescence relation is a strict partial order relation and subset of causal ordering.

When presenting algorithms a single thread of control is assumed. There-

fore each **upon/do** clause is assumed to be executed without interleaving with other processing. When several clauses are enabled, *i.e.*, their pre-condition is true, one of them is chosen non-deterministically to be executed. Fairness assumptions are presented in the text when applicable.

Algorithms use sets and queues as auxiliary data structures. The usual notation is used for sets in addition to procedures *add*(S, e) and *remove*(S, e), that insert or remove element e in the set variable S. Queues are used with procedures *addToTail*(Q, e), that inserts element e in the queue variable Q, and *remove*(Q, e), that removes element e from Q. Function *removeFirst*(Q) removes and returns the first element of Q. Function *merge*(Q_1, Q_2) returns a queue containing the union of elements of Q_1 and Q_2 and preserving their relative ordering. It is assumed that every two elements appearing in both parameters, do so in the same order. Set operations are used on a queue, denoting operations on the set of elements of the queue.

4.2 Semantically sender-based reliable multicast

4.2.1 Specification

Semantic reliability can be applied to a simple multicast protocol suited to information dissemination when there are no consistency requirements upon the failure of the sender. Semantically Sender-based Reliable Multicast (S-SRM), that relies on the correctness of the sender of each message, is defined by:

Semantic Sender-based Reliability: If a correct process multicasts a message m and there is a time after which no process multicasts m'' such that $m \sqsubset m''$, then all correct processes deliver some m' such that $m \sqsubseteq m'$.

Integrity: For every message m, every process delivers m at most once and only if m was previously multicast by some process.

FIFO Delivery: If a process multicasts a message m before it multicasts a message m', no process delivers m after delivering m'.

The intuitive notion that a message can be substituted by another that makes it obsolete is captured in the previous definition by the statement "deliver some m' such that $m \sqsubseteq m'$". In addition, if there is an infinite sequence of messages in which each message obsolete all its predecessors, the implementation may omit all of these messages. The possibility of omitting all messages that belong to an infinite sequence is captured by the statement "there is a time after which no process multicasts m'' such that $m \sqsubset m''$". It may seem awkward at first that such occurrence is allowed. However, it should be noted that the application, by defining the obsolescence relation, can prevent such sequences from occurring. Actually, the application can decide exactly which is the most appropriate length of any sequence of messages related by obsolescence. On the other hand, if the protocol was forced to deliver messages from a possibly infinite sequence (by omitting the statement above from the specifications), the protocol designer would be forced to make an arbitrary decision of which messages to choose from that infinite sequence (*e.g.*, one out of every k messages). It is clearly preferable to leave this decision to the application.

4.2.2 Algorithm

As the specification relies on the correctness of the sender process, the algorithm simply sends a message to each of the destinations upon multicast. As channels are reliable and FIFO, upon reception the message can be delivered. However, this does not allow any message to be purged and thus does not result in implementations that allow performance improvements due to semantic reliability.

Illustrating purging requires that buffering is made explicit. In the algorithm of Figure 4.1 this is done both in the sender with *to-send* and in the receiver with *to-deliver*. When a message is multicast ($t1$), it is placed in an

```
Initially:
    for all p: to-send[p] = empty queue
    to-deliver = empty queue

proc purge(Q) do
    while ∃ m, m' ∈ Q: m ⊏ m' do
        remove(Q, m)

t1 : upon multicast(m) do
    forall p ∈ destinations do
        addToTail(to-send[p], m)
        purge(to-send[p])

t2 : upon to-send[p]≠ ∅ do
        m ←removeFirst(to-send[p])
        send(m,p)

t3 : upon receive(m,p) do
        addToTail(to-deliver, m)
        purge(to-deliver)

t4 : upon to-deliver ≠ ∅ do
        m ←removeFirst(to-deliver)
        deliver(m)
```

Figure 4.1: Semantic Sender-based Reliable Multicast.

outgoing buffer *to-send[p]* for each destination process p. This enables $t2$ for each of the destinations, which being executed transmits the message. When a process receives a message ($t3$) it places it in the incoming buffer *to-deliver*. This enables $t4$ which eventually delivers it. This implies the following fairness assumptions:

- transitions $t2$ and $t4$ are weakly fair, *i.e.*, they cannot be enabled forever in a correct process without being eventually executed [Lam94];
- transition $t3$ is also weakly fair, according to the assumption of reliable channels;
- no fairness assumption is required on $t1$, as an application that never multicasts messages is still correct.

Notice that the algorithm does not otherwise specify the interleaving of transitions and thus how long or how many messages are stored in each buffer. This abstracts flow-control mechanisms that delay network transmission ($t2$) and delivery ($t4$) for bounded amounts of time. Purging happens when messages are added to a buffer in $t1$ and $t3$ by executing the *purge* procedure.

This searches for pairs of messages related by the obsolescence relation and removes the obsolete message thus avoiding it to be transmitted or delivered.

4.2.3 Correctness argument

The correctness of the algorithm is tightly related to the properties of point-to-point channels. The proof of integrity reduces to two simple to prove invariants: *i)* the set of messages in any *to-send*, *to-deliver* and in transit in point-to-point channels is a subset of messages multicast; *ii)* for any pair of processes p, q, sets of messages in *to-send*$[q]$ of p, in *to-deliver* of q, and in transit in the channel from p to q are disjoint. Both rely on point-to-point channels not creating or duplicating messages. The proof of FIFO Delivery is also a simple invariance proof: Consider the path from a sender p to a receiver q. No message in *to-send*$[q]$ of p is a predecessor of a message in the channel from p to q and none of these is a predecessor of a message in *to-deliver* of q. This relies also on the properties of FIFO channels and on the invariants used to prove Integrity.

The proof of Sender-based Reliability is as follows: Consider a message m multicast by a correct process p such that no m' with $m \sqsubset m'$ exists, and any correct destination process q. As no such m' exists, the message is never removed from *to-send* by procedure purge. This means that it will eventually be transmitted (as all predecessors are eventually purged or also transmitted) and received by q. Again, as no m' exists, the message is never removed from *to-deliver* by procedure purge. This means that it will eventually be delivered (as all predecessors are eventually purged or also transmitted).

4.3 Semantically reliable multicast

4.3.1 Specification

A semantically reliable protocol provides consistency guarantees even when a sender fails, thus being suited to be used for replication in fault tolerant systems. Semantically Reliable Multicast (S-RM), a protocol that enforces an agreement property despite the correctness of the sender, is defined by:

Semantic Validity: If a correct process multicasts a message m and there is a time after which no process multicasts m'' such that $m \sqsubset m''$, then it delivers some m' such that $m \sqsubseteq m'$.

Semantic Agreement: If a correct process delivers a message m and there is a time after which no process multicasts m'' such that $m \sqsubset m''$, then all correct processes deliver some m' such that $m \sqsubseteq m'$.

Integrity: For every message m, every process delivers m at most once and only if m was previously multicast by some process.

FIFO Delivery: If a process multicasts a message m before it multicasts a message m', no process delivers m after delivering m'.

Semantic FIFO Completeness: If a process multicasts a message m before it multicasts a message m' and there is a time after which no process multicasts m''' such that $m \sqsubset m'''$, no correct process delivers m' without delivering some m'' such that $m \sqsubseteq m''$.

This protocol differs from S-SRM by including agreement properties. Semantic Agreement is similar to the Agreement property in conventional reliable multicast and ensures that messages are delivered to either all or none of correct processes despite the failure of the sender. To understand the relevance of Semantic FIFO Completeness consider the following scenario in the information dissemination application of Section 2.3.1: a process multicasts m_1, m_2, m_3

such that only $m_1 \sqsubset m_3$ and then fails. If a correct process delivers only m_2 it would satisfy all properties of S-RM except Semantic FIFO Completeness and would result in a final state that never existed in the server. Notice that an observer process can temporarily exhibit this state, as long as it eventually delivers m_3. Semantic FIFO Completeness is thus a liveness property and is required in addition to the usual FIFO Delivery safety property. This makes liveness proofs more important in semantically reliable protocols.

Together, Semantic Agreement and Semantic FIFO Completeness properties ensure that the same prefix of non-obsolete messages are received by all correct processes. This agrees with the informal specification of the previous Section 3.4 and can be used, for instance, in the primary-backup replication scenario outlined.

Notice that when the obsolescence relation is empty, this specification reduces to conventional reliable multicast [HT94]. On the other hand, if every message makes all its predecessors obsolete, it results in an extension to multicast of *stubborn channels* [GOS98]. In between, various scenarios (such as those of Section 3.4) are possible and result in a generic protocol that is configured by the obsolescence relation.

4.3.2 Algorithm

Ensuring the agreement property in spite of the failure of the sender process requires that all processes are able to relay received messages to other processes as in reliable multicast [HT94]. This requires that each destination keeps track of messages already received in order not to deliver them more than once.

In addition, the protocol has to take additional precautions not to violate Semantic FIFO Completeness when purging obsolete messages. Consider the following scenario and sequence of events: a process p multicasts messages m_1, m_2, m_3 such that only $m_1 \sqsubset m_3$; p purges m_1 due to m_3; p sends m_2 that is

```
Initially:
    for all p: to-send[p] = empty queue
    for all p: received[p] = ∅
    to-deliver = empty queue
    delivered = empty queue

func new(Message m) =
    ∀ p: m ∉ received[p]

func safe(Message m) =
    |p: m' ∈ received[p]∧ m ⊑ m'| > f

proc purge_r(Q) do
    while ∃ m, m' ∈ Q: m ⊏ m' ∧ safe(m') do
        remove(Q, m)

proc purge_d(Q) do
    while ∃ m, m' ∈ Q: m ⊏ m' do
        remove(Q, m)

t1 : upon multicast(m) do
    addToTail(to-send[self], m)

t2 : upon to-send[p]≠ ∅ do
    m ←removeFirst(to-send[p])
    send(m,p)

t3 : upon receive(m,q) do
    if new(m) then
        forall p ∈ destinations: p ≠ self do
            addToTail(to-send[p], m)
        addToTail(to-deliver, m)
        purge_d(to-deliver)
    addTo(received[q], m)
    addTo(received[self], m)
    forall p ∈ destinations do
        purge_r(to-send[p])

t4 : upon to-deliver ≠ ∅ do
    m ←removeFirst(to-deliver)
    addToTail(delivered, m)
    deliver(m)
```

Figure 4.2: Semantically Reliable Multicast.

delivered by some process q; p crashes. Clearly, this sequence violates Semantic FIFO Completeness. The problem is that m_1 was purged before ensuring the eventual delivery of m_3. A message is guaranteed to be eventually delivered as soon as it has been received by $f+1$ processes, where f is the maximum number of processes that may fail. When this condition holds, we say that the message is *safe*. In the particular sequence above, violation of Semantic FIFO Completeness could be avoided if purging of m_1 was delayed until m_3 was known to be safe.

Both these issues are addressed by keeping track of which messages have been received from each process in *received[p]* and captured in functions *new*(m) and *safe*(m) in the algorithm presented in Figure 4.2. Notice that purging of the delivery queue does not need to be restricted to safe messages and thus

different procedures are used to purge the retransmission queues (*to-send*)and the delivery queue (*to-deliver*).

The algorithm works as follows: Upon multicast ($t1$) the message is queued and ($t2$) later sent only to the sender process itself. The message is then relayed to all destinations (except to the sender itself) and queued for delivery upon reception for the first time ($t3$). As in S-SRM, messages that are never purged are eventually delivered ($t4$). This implies the following fairness assumptions:

- transitions $t2$ and $t4$ are weakly fair, *i.e.*, they cannot be enabled forever in a correct process without being eventually executed [Lam94];
- transition $t3$ is also weakly fair, according to the assumption of reliable channels;
- no fairness assumption is required on $t1$, as an application that never multicasts messages is still correct.

Purging is performed only upon reception ($t3$) only if a message has been queued for delivery. The retransmission queue is purged regardless of no new message having been queued for retransmission, as an existing message might have become safe and thus made purging possible. Notice that there is no point in purging *to-send* in $t1$, as the new message is guaranteed not to be safe yet as it is still waiting to be sent to the sender itself.

4.3.3 Correctness argument

Safety properties (Integrity and FIFO Delivery) are similar to their counterparts in reliable multicast. The correctness of liveness properties of S-RM is not as simple as for S-SRM. This justifies a more detailed proof of liveness properties in Appendix A.

4.4 Semantically view synchronous multicast

4.4.1 Specification

To describe the combination of view synchrony with semantic reliability we consider a simplified group membership service in which processes can only leave the group. The kind of events that may lead to a view change are not relevant to the definition of Semantic View Synchrony. Examples of possible causes for triggering a view change to remove a process from the group are the occurrence of failure suspicions [LH99], the lack of available buffer space at one or more processes [CBDS01] and simply the existence of processes that voluntarily want to leave. The properties for Semantically View Synchronous Multicast (S-VSM) are:

Semantic View Synchrony: If a process p installs two consecutive views v_i and v_{i+1} and delivers a message m in view v_i, then all other processes installing both v_i and v_{i+1} deliver some m' such that $m \sqsubseteq m'$ before installing view v_{i+1}.

Integrity: For every message m, every process delivers m at most once and only if m was previously multicast by some process.

FIFO Delivery: If a process multicasts a message m before it multicasts a message m', no process delivers m after delivering m'.

Semantic FIFO View Completeness: If a process multicasts a message m before it multicasts a message m' and a process p installs two consecutive views v_i and v_{i+1}, and delivers message m' in view v_i, then q delivers some m'', such that $m \sqsubseteq m''$, before installing view v_{i+1}.

The Semantic FIFO View Completeness property relaxes the traditional Reliable FIFO properties [CKV01]. Given a sequence of messages multicast by a process, this ensures that upon view installation only obsolete predecessors

of the last message delivered can be omitted. With Semantic View Synchrony every two processes installing two consecutive views v_i and v_{i+1} do not necessarily deliver the same sequence of messages, thus being weaker than View Synchrony, but they are ensured to deliver (at least) the same sequence of messages that have not been made obsolete by subsequent messages up to view v_{i+1}. For instance, process p_1 may deliver in view v_i messages m_1 and m_2, such that $m_1 \sqsubset m_2$, and process p_2 may only deliver m_2 in the same view. If no messages m, m' exist such that $m \sqsubset m'$, Semantic View Synchrony reduces to conventional View Synchrony. This makes it more general as different concrete semantics, including View Synchrony, can be obtained by defining an appropriate obsolescence relation.

4.4.2 Algorithm

Interestingly, S-VSM can be obtained simply by adapting an existing view synchronous protocol to include purging of obsolete messages at the appropriate steps. It is hence possible to derive S-VSM implementations to different systems models, by adapting different view synchronous implementations. The purpose of this section is not to re-invent view synchronous protocols, since these have been extensively studied in the literature [HS95]. However, we do want to illustrate what changes are needed to accommodate S-VSM. To do so, we opted to adapt a protocol designed to run on asynchronous systems augmented with a failure detector, that only allows processes to leave the group and that uses consensus as a building block [GS01]. The algorithm is depicted in Figures 4.3 and 4.4.

Each process in the group keeps a variable *cv* with the most recent view, a boolean variable *blocked* that is used to prevent the reception and transmission of new messages during the view change protocol, and two queues of messages ordered by FIFO: *to-deliver* and *delivered*. When messages are received they are inserted in the *to-deliver* queue for delivery to the application. A mes-

sage m in the *to-deliver* queue may be purged if a message m' such that $m \sqsubseteq m'$ is received in the same view. This is modeled by the *purge* procedure. Delivery of a message m implies removing it from the head of *to-deliver* and adding it to the tail of the *delivered* queue ($t4$).

Figure 4.3 depicts the part of the algorithm that deals with multicast and delivery of messages between installation of views. Data messages can only be multicast if the group is not blocked ($t1$). A multicast message is tagged with the current view and sent to all other processes in the view. A data message is denoted DATA*(v, m)*, where v is the view in which it is sent. Additional tags are used for views and two control messages whose purpose is explained in the following paragraphs.

Upon multicast, the message is also inserted in the *to-deliver* queue of the sender. This will ensure that if the sender participates in the next view, all the messages it has sent will be delivered in the current view. Data messages are only accepted if the recipient is still in the view they were sent and if the group is not blocked ($t2$). As before, received messages are added to the *to-deliver* queue of the recipient.

Figure 4.4 depicts the part of the algorithm that deals with view installation. The installation of a new view is triggered by an external event. In response to this event, the initiator of the view change simply disseminates a INIT control message to all group members ($t4$). Upon the reception of the first INIT message, a process forwards the INIT to all other members, ensuring that all correct processes initiate the view change ($t5$). Additionally, each process computes the sequence of messages it has accepted to deliver in the current view and sends this sequence to all other processes in a PREC control message. These sets are collected by all correct processes in the *global-pred* set ($t6$). The set of processes from which the PREC message has been received for the current view is maintained in the variable *pred-received*. When *pred-received* includes all processes from the current view that are not suspected, and this set contains a majority of processes, a new view as well as the sequence of

```
Initially:
    to-deliver = empty queue
    delivered = empty queue
    cv = (0, ids of all processes)
    blocked = false

proc purge(Q) do
    while ∃ DATA(v, m), DATA(v', m') ∈ Q: v=v' ∧ m ⊏ m' do
        remove(Q, DATA(v,m))

t1 : upon multicast(m) ∧¬ blocked ∧ self ∈ memb(cv) do
    addToTail(to-deliver, DATA(cv, m))
    forall p ∈ memb(cv): p ≠ self do
        send(DATA(cv, m),p)
    purge(to-deliver)

t2 : upon receive(DATA(v, m), p): v = cv ∧¬ blocked do
    if ∄ DATA(v',m') ∈ (to-deliver ∪ delivered): v=v' ∧ m ⊑ m' do
        addToTail(to-deliver, DATA(v, m))
        purge(to-deliver)

t3 : upon to-deliver ≠ ∅ do
    m ← removeFirst(to-deliver)
    addToTail(delivered, m)
    if m = DATA(v,d) then
        deliver(d)
    else if m = VIEW(v) then
        cv ← v
        blocked ← false
        install(v)
```

Figure 4.3: Semantic View Synchrony: multicast operations.

Initially:
    global-pred = empty queue
    pred-received = $\emptyset$
    leave = $\emptyset$

$t4$: **upon** trigger-view-change (l) **do**
    forall p $\in$ memb(cv) **do**
        send(INIT(cv, l),p)

$t5$: **upon** receive(INIT(v, l),p): v = cv $\wedge\neg$ blocked **do**
    if p $\neq$ self **do**
        forall p $\in$ memb(cv) **do**
            send(INIT(v, l),p)
    blocked $\leftarrow$ true
    leave $\leftarrow$ l $\cap$ memb(cv)
    local-pred $\leftarrow$ { DATA(v, m) $\in$ (delivered $\cup$ to-deliver): v = cv }
    forall p $\in$ memb(cv) **do**
        send(PRED, cv, local-pred],p)

$t6$: **upon** receive(PRED(v, P),p): v = cv **do**
    global-pred $\leftarrow$ merge(global-pred,P)
    pred-received $\leftarrow$ pred-received $\cup$ {p}

$t7$: **upon** $|\text{pred}| > \frac{|\text{memb(cv)}|}{2} \wedge \forall$ p $\in$ memb(cv): suspects(p) $\vee$ p $\in$ pred **do**
    proposed-view $\leftarrow$ (id(cv)+1, pred-received\leave)
    (next-view, pred-view) $\leftarrow$ consensus(cv, (proposed-view, global-pred))
    if self $\in$ memb(next-view) **do**
        while pred-view $\neq \emptyset$ **do**
            m $\leftarrow$ removeFirst(pred-view)
            if m $\notin$ (to-deliver $\cup$ delivered) **do**
                addToTail(to-deliver, m)
        addToTail(to-deliver, VIEW(next-view))
        purge(to-deliver)

Figure 4.4: Semantic View Synchrony: view change.

messages to be delivered in the current view are proposed for consensus ($t7$). The proposed view corresponds to the *pred-received* set (minus the processes leaving the group in variable *leave* that is given as an input parameter to the view change procedure).

After consensus returns, the agreed sequence of messages to be delivered in the current view is added to the *to-deliver* queue followed by the agreed next view control message. These messages are delivered ($t3$), eventually updating the current view and unblocking the process.

4.4.3 Correctness argument

When addressing the correctness of the algorithm we focus on Semantic View Synchrony and Semantic FIFO View Completeness. The reason for this is that these are the properties that differ from those found on VS algorithms and thus reflect the impact of purging obsolete messages.

The original view synchrony protocol, obtained from Figures 4.3 and 4.4 without the shaded lines or with an empty obsolescence relation, implements conventional view synchronous multicast [GS01]. From this we can derive the correctness of the implementation of S-VSM considering the following fact: the purge operation never discards maximal elements (according to the obsolescence relation $\sqsubset$) of the set of messages delivered by some process prior to installing a given view. If a process participates in view v_{i+1} and purges some message m, then there is some m' in *to-deliver* $\cup$ *delivered* such that $m \sqsubset m'$ that would be included in the *pred-view* set decided for v_{i+1} and thus m would not be maximal. For any message m delivered by some process installing both v_i and v_{i+1}, either (i) m is maximal in the set of messages and thus is never purged and as in the original algorithm delivered by all processes before installing v_{i+1} or (ii) m is not maximal and there is some m' such that $m \sqsubset m'$ that is maximal.

The argument for Semantic FIFO View Completeness is similar. As chan-

nels are reliable and FIFO, it can easily be shown that without ever purging messages, *to-deliver* $\cup$ *delivered* contain always complete prefixes of sequences of messages multicast by each sender. The subset of maximal elements (according to the obsolescence relation $\sqsubset$), that is guaranteed to be maintained by purging is sufficient to ensure the desired property.

4.5 Uniform agreement

The protocol definitions presented are not uniform, *i.e.*, properties hold only for correct processes. Uniform agreement is important in preventing contamination [GT91], where the effect of a message delivered only to an incorrect process before it crashes becomes visible to other processes. An uniform version of agreement can be defined as:

Semantic Uniform Agreement: If a process delivers a message m and there is a time after which no process multicasts m'' such that $m \sqsubset m''$, then all correct processes deliver some m' such that $m \sqsubseteq m'$.

The transformation from non-uniform to uniform agreement in reliable multicast can be done by delaying the delivery of a message m until it has been received from at least $f+1$ processes. In a semantically reliable protocol, it suffices to receive from each of a majority of processes some m' such that $m \sqsubseteq m'$. This is already expressed as *safe*(m'). Therefore, the same mechanism used to detect safety for purging can be used to ensure uniform agreement.

4.6 Causal and total order

Although we have presented only FIFO order, one can also combine semantic reliability with causal and total order constraints. For causal order this involves defining a suitable completeness property:

Causal Delivery: If a message m causally precedes a message m', no process delivers m after delivering m'.

Semantic Causal Completeness: If a message m causally precedes a message m' and there is a time after which no process multicasts m''' such that $m \sqsubset m'''$, no correct process delivers m' without delivering some m'' such that $m \sqsubseteq m''$.

The implementation of Semantic Causal Completeness is similar to that of Semantic FIFO Completeness and requires delaying purging until a message is safe. It is however not possible to ensure multicast ordering solely based on the order of point-to-point channels. This requires that a message received is stored in an intermediate buffer until all predecessors are queued or known to be purged. This is further discussed in Appendix A.

The definition of Uniform Total Order given in Section 2.2.4 remains valid and can be used directly. A protocol for total order can be layered on top of S-RM in the same fashion as proposed for reliable multicast [CT96]. Briefly, messages are reliably multicast and upon delivery from reliable multicast, messages are temporarily bufferd such that in each process:

1. the set of messages stored is proposed to consensus;

2. upon decision, the set of messages decided is delivered by some deterministic order and removed from the set of pending messages;

3. repeat from step 1 with remaining and newly delivered messages.

For semantic reliability, upon each decision, one purges all pending messages that are known to be obsolete, either by other pending messages or by messages appearing in the decided set. Notice that by using uniform consensus, this protocol also ensures the Semantic Uniform Agreement property.

4.7 Summary

In this chapter we have introduced the definitions of protocols that constitute a semantically reliable group communication toolkit, namely, reliable and view synchronous multicast combined with FIFO, causal and total order. To provide the same convenience for fault tolerant programming, an additional liveness property is required when ordering messages.

Interestingly, algorithms for semantic reliability can be derived from the algorithms for conventional reliability by adding purging operations: if two related messages are found within the same buffer, the one that is obsolete can be purged. The only additional limitation is the additional liveness property, which requires the collection of acknowledgments before purging messages in some situations.

Chapter 5

Performance Evaluation

The performance of semantically reliable multicast protocols is evaluated using three complementary techniques: a simple analytical model, a simulation model and a prototype implementation. The results obtained relate the performance of semantic reliability with the application profile and system configuration parameters. The prototype implementation allows also to evaluate the overhead imposed by additional protocol mechanisms required for semantic reliability.

5.1 Performance models

A simple analytical model allows the application designer to assess the expected purging rate that can be observed with a concrete obsolescence pattern and system configuration, and thus the adequacy of semantic reliability for a specific application. A simulation model allows estimating the performance of the protocol in complex settings before it is actually implemented. This validates the analytical model and allows the protocol developer to evaluate the interaction of semantic reliability and other protocol mechanisms.

We consider a simplified system model constituted by a single sender, a fast receiver and a slow receiver (see Figure 5.1). The sender produces mes-

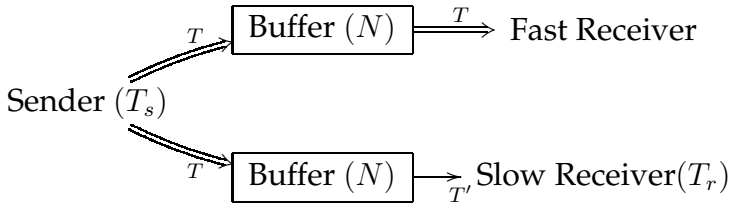


Figure 5.1: Simplified system model.

sages at a rate up to T_s. For each receiver, messages are placed in a buffer with capacity for N messages. If a message cannot be inserted in one of the buffers, the sender blocks until buffer space becomes available. A fast receiver removes messages from its buffer as soon as they become available, thus never making the sender block. On the other hand, the slow receiver removes messages from its buffer at rate T', which is at most T_r. Considering $T_r < T_s$, the slow receiver's buffer eventually fills up. This blocks the receiver thus impacting T, the rate at which messages are effectively produced by the sender. If no messages are ever discarded in buffers, as in a reliable protocol, then $T = T'$.

5.1.1 Analytical

With semantic reliability, the protocol can purge the buffer of obsolete messages to store arriving messages. Nevertheless, if the system remains overloaded for a long period, the buffer will eventually be filled just with unrelated messages. Therefore, new messages can only be accepted if they obsolete one of the messages in the buffer.

The estimation of performance thus depends on knowing the distance in the input stream between related messages. Unless obsolescence is strictly periodic, this is a random variable. Let D be the distance between each message and the latest message it makes obsolete, and $f(x) = P(D = x)$ the probability mass function of D. Value $f(0)$ is defined to be the probability of not existing any obsolete predecessor message.

The probability of a message being obsoleted by a new message is thus given by $R_* = \sum_{x \geq 1} f(x)$, that is an estimate of the maximum ratio of messages that can be purged by the protocol under continued congestion. However, this is not a good estimate of how the protocol behaves, as it implicitly assumes an unbounded amount of previous buffered messages.

Knowing that when the system is congested buffers are full, a more reasonable assumption is to consider that the buffer size determines the maximum distance between two related messages such that one of them can be discarded. Making the simplifying assumption that the buffer holds messages sent immediately before, the total probability of an obsoleted predecessor existing in the buffer is thus $R_N = \sum_{1 \leq x \leq N} f(x)$, where N is the maximum number of messages buffered for each receiver. This gives an estimate of the ratio of messages that can be purged by the protocol under continued congestion. Using R_N and given maximum sender and receiver throughputs T_s and T_r, it is possible to derive the effective throughputs T, departing from the sender and being consumed by a fast receiver, and T' (see Figure 5.1):

$$T = \min(T_s, \frac{T_r}{1 - R_N})$$
$$T' = \min(T, T_r)$$

Naturally, if probability accumulates at low values of distance, *i.e.*, if the probability of a message being made obsolete by a close subsequent message is high, the purging procedure is very effective. On the other hand, if the distance is large, it is likely that the buffers become exhausted before any message has the chance to become obsolete. It is also clear that, for the same obsolescence distribution, the algorithm performs best for larger buffer sizes.

5.1.2 Simulation

The analytical model does not take into consideration several issues that may affect the efficiency of the algorithm. To start with, it does not consider the

effect of the purging procedure itself in the content of the buffer, which means that even if exactly N messages are stored, they are likely not to be the last N messages. Furthermore, existing networks are not fully reliable and may deliver packets out of order. Thus, the actual distribution of messages in the recipient's buffers is even more unpredictable than considered above, where we assume that all messages are received in FIFO order. Thus the buffer might hold any N previous messages or even some subsequent messages. Additionally, in a real system we do not have a single buffer for each pair of sender-receiver nodes. Instead, we have two partially overlapping buffers, one at the sender and the other at the recipient, where purging may be applied. Naturally, if obsolete messages are purged in the sender's buffer, there is the possibility that some obsolete information never reaches recipients. On the other hand, there is less load imposed downstream.

By using a simulation model, we can confirm the validity of the analytical model despite its simplicity and explore the impact of system parameters in the performance of purging in more complex models. A discrete-event simulation model [Jai91] works by keeping a queue of events ordered by their scheduled time. The simulation progresses by removing and executing the event in the head of the queue. The scheduled time of the event is considered the current time during the execution. Executing an event handler may change the state of the model and schedule further events to a subsequent time. Simulation terminates when the event queue is empty or the time reaches a pre-established instant.

The system state is composed by a pair of FIFO buffers with configurable size N, one for the fast and one for the slow receiver. Events are periodically scheduled to produce and consume messages according to at most T_s and T_r. The obsolescence relation can either be generated randomly or replay a profile obtained from a real application. This obsolescence relation is represented using k-enumeration, with k defined as a parameter. The following variations of the model are used to study the impact of implementation mechanisms:

- Purging can be performed as soon as possible despite existing free space (eager purging), or only when necessary to accommodate a message arriving to a full buffer (lazy purging).

- Each buffer can be split in two sections, modeling buffering in both ends of a network connection. A message in each section can only be purged if it is made obsolete by another message arriving or already in the same section.

- A delay is imposed in each message until it can be used for purging. This models the effect of gathering acknowledgments before purging to satisfy completeness properties in ordered protocols.

The simulation logs the time when each messages is produced, enters each buffer, leaves each buffer and is consumed. Message rates, purging rate, average usage of buffers and the delay caused by queuing are can then be computed.

5.2 Prototype implementation

A prototype implementation of semantically reliable multicast with static group membership allows experimental evaluation of semantic reliability, in particular, of its impact on throughput stability and of the overhead of the additional mechanisms required to deal with semantic reliability.

The prototype is implemented using the C++ language and the ACE framework [SBS93] as an operating system abstraction layer. Specifically, it uses a Reactor object for scheduling and datagram sockets for communication, both point-to-point and multicast. The protocol code is event-driven and executes in a single thread. Events can be triggered by arrival of messages from the application or the network, by timeouts or explicitly queued by event handlers. Event handlers are statically associated with event types.

Application code runs in a separate thread. The composition of the group is fixed and must be supplied to the protocol during initialization of each node. After that, a message is multicast by executing a blocking `write()` operation on the protocol. Delivery is initiated by the application that invokes a blocking `read()` operation on the protocol.

Although the protocol architecture is monolithic, it is possible to conditionally compile protocol code to suppress some features, namely, buffering of each message by processes other than the sender, FIFO order and purging. This allows the impact of purging to be compared in protocols that are otherwise identical.

5.2.1 Retransmission and flow-control

When a message is multicast it is optimistically disseminated using IP multicast [DC90] and then buffered. An optional upper bound on the bandwidth consumed by multicast can be imposed to avoid congesting the network. Reliability is ensured by a receiver initiated mechanism [PTK94, Cla82]. Each receiver keeps a queue of messages discovered to be missing which it requests using negative acknowledgment messages. To avoid congesting the network, retransmission requests are controlled by a variable window that uses the TCP/IP algorithm [Jac88].

Garbage collection of retransmission buffers is based on a scalable stability tracking algorithm [Guo98]. This algorithm uses gossiping to determine which is the last message that has been received by all processes. Intermediate control messages of this protocol are used by receivers as hints in the discovery that the last message sent by some process has been lost.

Flow control is performed by imposing a limit on the total number of messages buffered (*i.e.*, for retransmission and for delivery) at each process. Therefore, if a process is consuming messages slower than they arrive, messages accumulate in its buffers. Eventually this process ceases to accept further data

messages from the network, making the stability tracking algorithm block in the latest message stored by such a process. Being unable to remove messages from its buffers, the sender protocol ceases to accept further messages from the application. Care is taken not to cause deadlocks when FIFO order is in use.

5.2.2 Purging

The obsolescence relation is represented using the k-enumeration technique and supplied to the protocol upon multicast as an additional parameter of the `write()` operation. Upon the arrival of a new message to a buffer, it is determined if any existing messages become obsolete and can later be purged. This is determined by storing and updating an index of the message buffer. By representing the obsolescence relation with a bitmap in the message, this index reduces to maintaining the latest messages stored in the buffer from each sender as a bitmap too. Upon arrival, a logical "and" of both bitmaps (after the appropriate shift operations) is performed. Each remaining set bit represents a message that can be purged from the buffer. Notice that iterating the resulting bitmap is reasonably fast as this bitmap is small. This suffices for purging the delivery queue in any of the semantically reliable protocols.

Purging the retransmission buffer involves an additional step. It may happen that a message is purged from all processes capable of retransmitting it before it has been delivered to all destinations. Processes that have not received a purged message must realize that this message can be removed from the negative acknowledgment queue and skipped. The mechanism used to do this is the stability tracking algorithm. If a process purges a message from its retransmission queue, it fakes the stability of that message by unilaterally declaring it received by all processes. Eventually, processes that have not received it discover that it has became stable, regardless of not having been locally received, and therefore conclude that the message is not missing but has

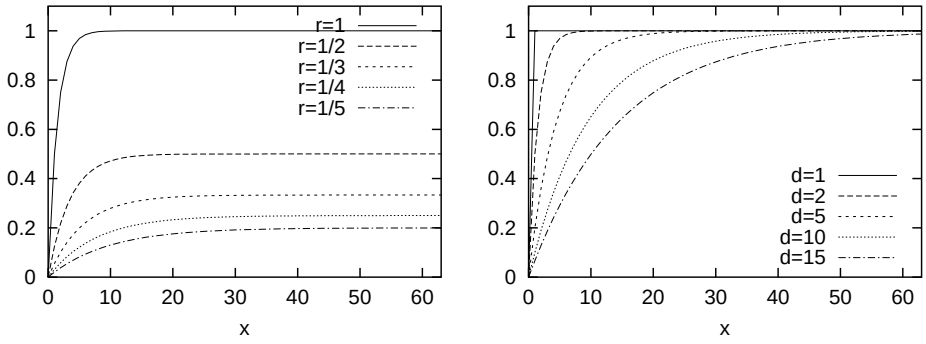
been purged and should be skipped.

Determining which messages are safe to be used for purging in FIFO protocols is achieved by running a global safety tracking algorithm, that determines which messages have been queued for delivery by $f + 1$ processes and delaying purging accordingly. This algorithm is a modified version of the stability tracking algorithm [Guo98] requiring only $f + 1$ votes (instead of n) to make progress.

5.2.3 Centralized simulation

The evaluation of the prototype implementation can be improved by using a centralized simulation model [AC97]. This combines a real implementation of the protocol code with discrete-event simulation models of the application and the network and has been shown to accurately reproduce timing properties of real systems. In detail, centralized simulation allows single-threaded event-driven code to be run side-by-side with simulated system components: The execution of each event handler is timed using a profiling timer and the result used to update a simulated timeline.

By running all processes under control of the centralized simulation runtime within a single workstation, it becomes possible to perform observations that depend on a centralized clock. By stopping the clock, it is also possible to perform detailed accounting of various system parameters during runtime without disturbing the results. Centralized simulation has also simplified testing and debugging of the protocol implementation, by allowing automation of regression tests with fault-injection. This configuration of the model has been validated both by micro-benchmarks for individual parameters (*e.g.*, overhead of the kernel network stack and scheduling latency) and by comparing results of protocol executions with results of the real system when possible (*e.g.*, distribution of round-trip times).



(a) Varying r with $d = 1$

(b) Varying d with $r = 1$

Figure 5.2: Plots of obsolescence distribution.

5.3 Experimental conditions

5.3.1 Traffic characterization

To exercise system models and the prototype implementation we have selected a pattern of message obsolescence consisting of two distinct types of messages: *i) independent* messages that do not make other messages obsolete and that are not made obsolete by any other message; and *ii) overwrite* messages that obsolete their predecessors and are made obsolete by their successor with a given probability. The distribution is characterized as follows:

$$f_{r,d}(x) = \begin{cases} 1 - r & \Leftarrow x = 0 \\ r(1 - \frac{r}{d})^{x-1} \frac{r}{d} & \Leftarrow x > 0 \end{cases}$$

This distribution is interesting because it is easily generated and because parameters r and d directly determine the characteristics of the traffic. The parameter r models the relative distribution of independent and overwrite messages: On the average, a ratio r of messages has overwrite semantics. Thus, r directly establishes an absolute upper bound on purging. As shown in Fig-

ure 5.2(a), $R_N \leq r$ and except for very small values of N, $R_N = r$. The parameter d represents the diversity of overwrite messages, dictating the probability of two overwrite messages being related and thus sensitivity to buffer size N. As shown Figure 5.2(b), the value of N required for R_N to approach 1 is increasingly larger with d. With this distribution we can explore boundary conditions that limit the performance of our protocol.

Simulations and prototype executions shown in the following section use traffic generated with constant intervals. When using the prototype implementation, each message carries a payload of 4 bytes, except for measurements of Figures 2.6(c) and 5.10(c), when it is 1000 bytes.

5.3.2 Performance perturbations

All of our models and the prototype allow setting $T_r < T_s$. This is used as the main performance perturbation scenario as it directly implies that messages are discarded to avoid blocking the sender. It models mainly performance perturbations during message processing by the application.

The centralized simulation model allows a wider range of perturbations to be injected, specifically, by making a process sleep by a fixed amount of time each second, by delaying or by dropping network packets. Temporarily blocking a process has an immediate impact in throughput as it halts all protocol mechanisms, including stability tracking [BHO+99]. Network perturbations affect both incoming and outgoing packets of a single process. Packet loss occurs according to available bandwidth. This requires retransmission of data and also delays stability. Packet delays are constant and impact mostly stability detection.

5.3.3 Environment

Measurements of the prototype implementation were obtained with a network of 3 Pentium III/1 GHz workstations over a switched 100 Mbits Ethernet. One of the workstations is used as the sender. Another is used as the slow receiver, by sleeping an amount of time between deliveries. The sender and the third receiver do not introduce additional delays between deliveries, therefore consuming messages as soon as they are available. Unless otherwise noted the protocols used are FIFO Reliable Multicast [HT94] (reliable) and S-RM (semantic). All throughput measurements are performed at the sender. Initial measurements in each run are discarded to obtain results only after the system is stationary.

The operating system used in all workstations is RedHat Linux 7.1 and the protocol is compiled using the default GNU C++ 2.96 compiler with optimization option -O2. As the ACE toolkit uses the `select()` system call for timing in Linux, the clock has a 10 ms resolution that limits the granularity of the sample. For instance, we used 10 ms as the period of the sender. Nevertheless, the low resolution of the timer used prevents obtaining measurements for higher message rates.

By timing the execution of real code in centralized simulation, the performance of the host processor determines the performance of each simulated processor. We used the same 1GHz workstation for measurements. The model used for the network assumes 10 μs delay for each pass of the UDP/IP stack within the operating system (as measured in Linux) and a simulated 100 Mbits full duplex switched network. Applications do not consume processor time. We have not simulated the 10 ms granularity of Linux timers, allowing us to obtain results other than for multiples of 10 ms. On the other hand, we modeled the scheduling latency of the operating system in order to be able to realistically reproduce round-trip measurements.

The prototype protocol is configured as follows, both for execution and

centralized simulation:

- maximum buffer size of 40 messages, regardless of their size;
- both the stability and safety detection protocols are configured with a period of 30 ms and a fanout of 3.

This configuration is tuned for high throughput with very small buffers on a high bandwidth and low latency network. For other networks, a larger buffers and period should be used [Guo98].

5.4 Results

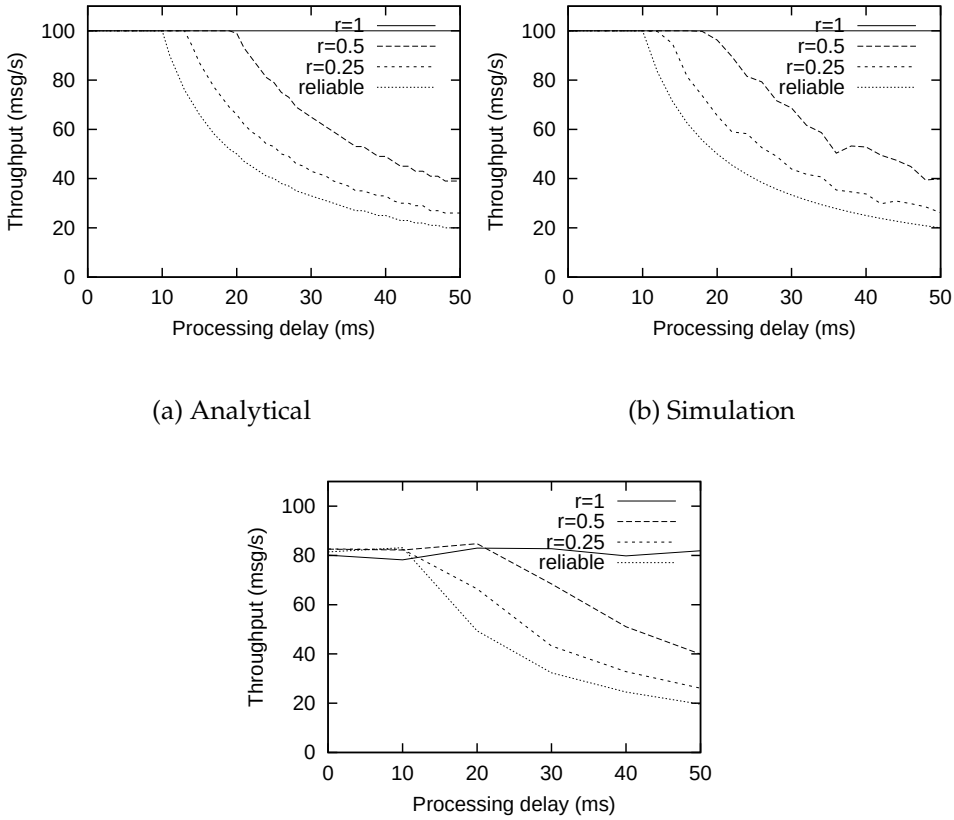
5.4.1 Purging efficiency

In this section we study the performance of purging based on message obsolescence traffic and system parameters. Whenever possible, the values obtained from the analytical model are compared with the simulation results and with the data collected from the implementation.

The effectiveness of purging protocol buffers, introduced by semantically reliable protocols, is measured mostly by the ability to accommodate a slower receiver within the group without disturbing the sender. This allows the resources of fast receivers to be fully used.

Figure 5.3 presents the sustained incoming throughput (T_s) as a function of the processing delay of a single slow receiver. When purging is not applied, processing delays larger than 10 ms prevent a delivery throughput of more than 100 msg/s, thereby reducing the input that can be accepted. For instance, with 20 ms only 50 msg/s are accepted.

By generating traffic with parameter $d = 1$ and a sufficiently large buffer size, parameter r directly determines the amount of traffic that can be purged. Using a semantically reliable protocol we observe that:



(a) Analytical

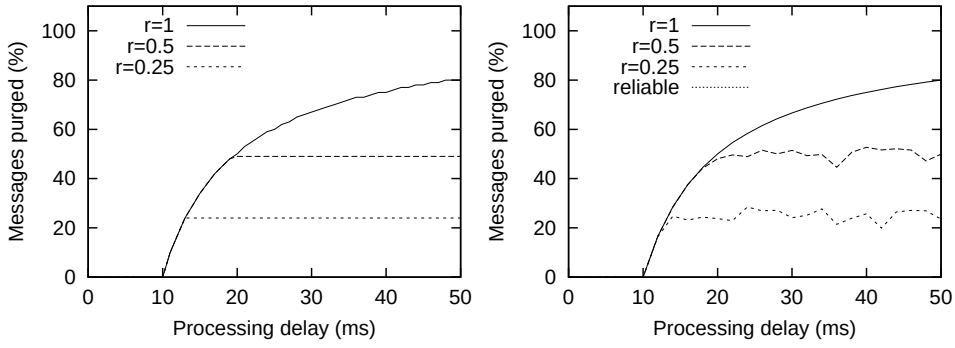
(b) Simulation

(c) Implementation

Figure 5.3: Throughput with $d = 1$, $N = 20$ and variable r.

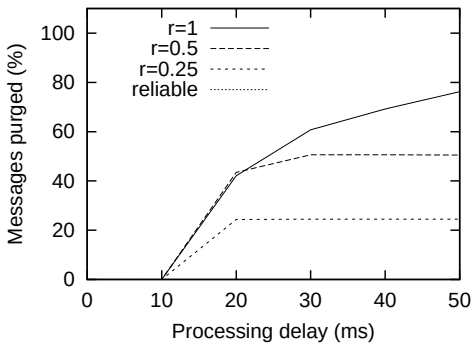
- When the amount of messages that can be purged is enough to accommodate the difference between T_s and T_r, the sender is undisturbed. For instance, with $r = 0.5$ half of the messages eventually become obsolete and can be purged. Therefore, the receiver can exhibit up to twice the delay (20 ms) without disturbing the sender.

- When the amount of messages that can be purged is not enough to accommodate the difference between T_s and T_r, such as with $r = 0.25$ and a delay of 20 ms, although the sender is disturbed the input allowed is still higher than without purging (75 msg/s versus 50 msg/s).



(a) Analytical

(b) Simulation

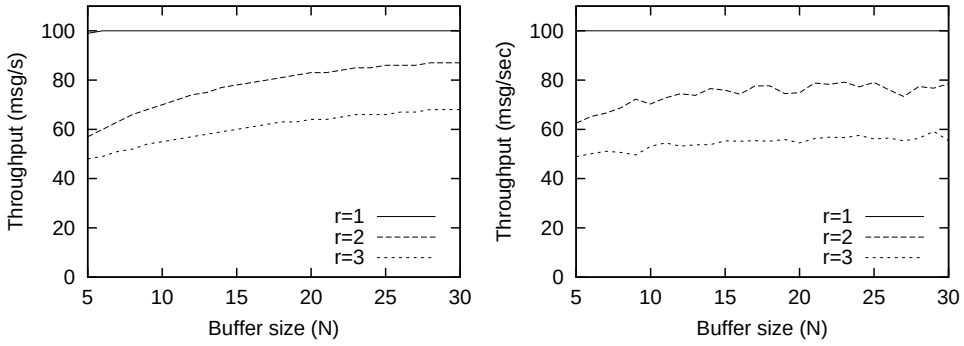


(c) Implementation

Figure 5.4: Messages purged with $d = 1$, $N = 20$ and variable r.

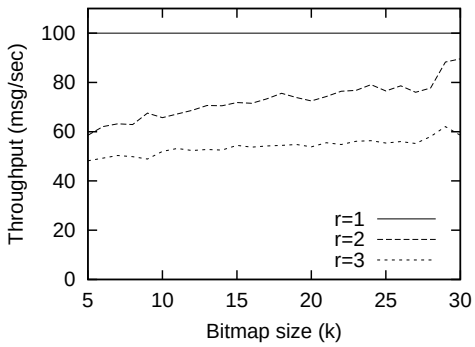
These results are explained by the amount of messages that are purged and thus are not delivered to the slower receiver as shown in Figure 5.4. These results confirm that the maximum expected purging rate and consequent improvement in throughput can in fact be observed in practice.

Notice that with the prototype implementation it has not been possible to achieve an input of 100 msg/s. This happens because the test application tries to sleep for 10 ms between sending messages. However, due to the lack of accuracy of the operating system timer, it is often scheduled later, therefore reducing the input rate. This also means that measurements were obtained only



(a) Analytical

(b) Simulation (buffer size)



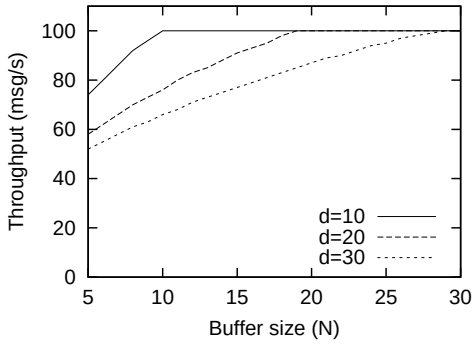
(c) Simulation (parameter k)

Figure 5.5: Buffer size sensitivity to r ($d = 5, T_r = 40$).

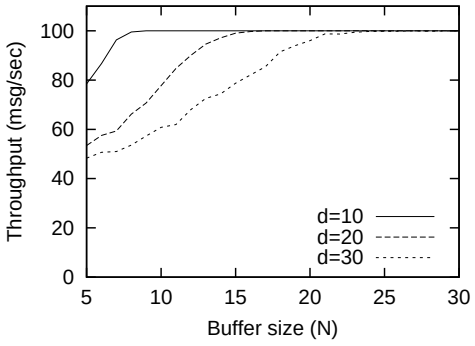
for delays multiple of 10 ms that reduces the detail of Figures 5.3(c) and 5.4(c).

Buffer and bitmap sizes (N and k) The possibility of purging messages depends on recognizing pairs of related messages within a buffer. This is affected by the size of the buffer, the representation of the obsolescence relation and the characteristics of the traffic. Using simulation we can easily observe the behavior of the protocol when such parameters are varied.

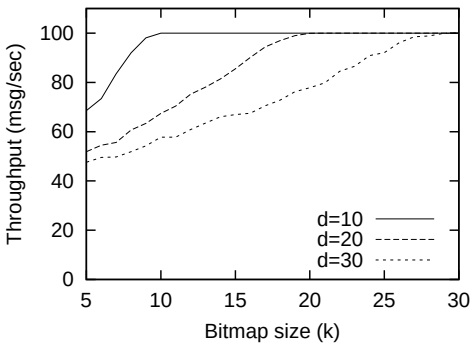
Figure 5.5 shows the impact of varying buffer size N and obsolescence representation parameter k with several values of r and a low value for d.



(a) Analytical



(b) Simulation (buffer size)

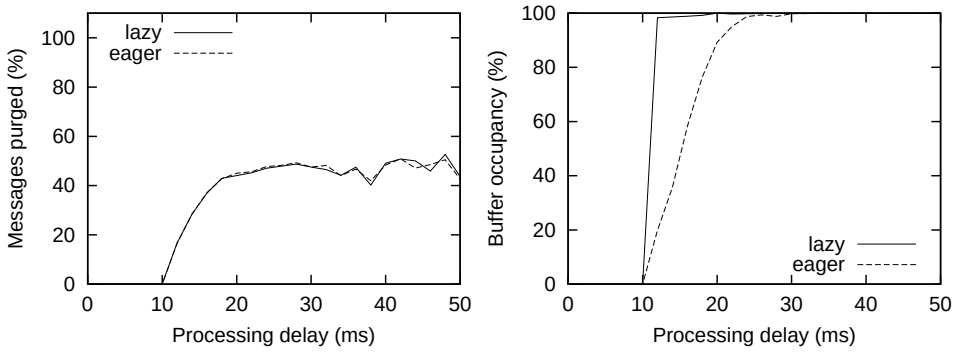


(c) Simulation (parameter k)

Figure 5.6: Buffer size sensitivity to d ($r = 1, T_r = 40$).

This makes the ratio of independent messages the limiting factor. Notice that the analytical model is somewhat optimistic. This happens because measurements were taken after the system is congested for a long time which means that buffers fill up with messages that never become obsolete.

On the other hand, if the limiting factor is d, the diversity of traffic, all messages eventually become obsolete although related pairs of messages are far apart. As shown in Figure 5.6 the analytical model illustrates how performance is degraded when parameter k of the obsolescence representation is too low. The analytical model is however to pessimistic when describing the



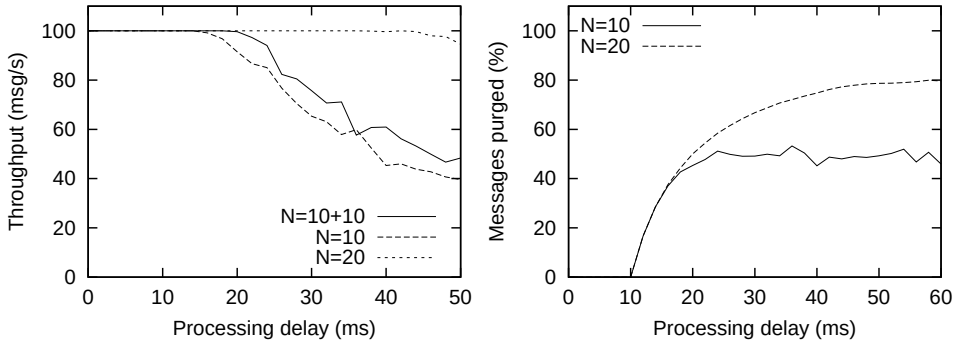
(a) Messages purged (b) Buffer occupancy

Figure 5.7: Comparison of purging strategies using simulation ($r = 0.5, d = 5$).

impact of a small buffer. This happens because purging makes related pairs of messages closer after purging others in between.

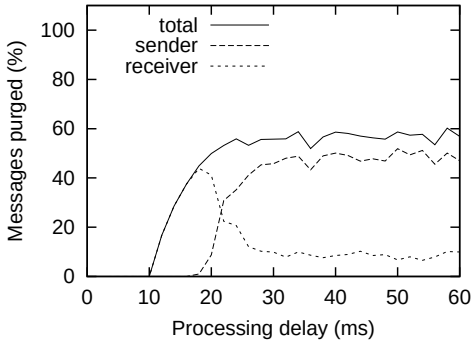
Eager and lazy purging In the analytical model we have also assumed that purging happens only when the buffer is full, thereby yielding a constant buffer size. This is a *lazy* purging strategy. In contrast, an *eager* purging strategy where purging is always applied can also be considered. As messages are delivered immediately, no messages are purged in the path to a fast receiver or when the system is not congested (Figure 5.7(a)). However, buffer usage while purging is effective is lower (Figure 5.7(b)). Therefore eager purging results in better latency and better response to short congestion periods without otherwise impacting performance. As searching for obsolete messages does not represent a major overhead we consider only eager purging.

Single and split buffer When the system is not congested, buffers at the sender and at receivers contain approximately the same messages, waiting to be garbage collected. When the system is congested, buffers of slow receivers are eventually filled up with older messages waiting for delivery, while fast



(a) Throughput

(b) Purging (single buffer)



(c) Purging (split buffer)

Figure 5.8: Buffer $N = 20$ compared with $N = 10 + 10$ ($r = 1, d = 20$).

receivers and the sender hold newer messages waiting to be transmitted to slower receivers. This raises the issue of being able to use twice the buffer size for purging. We now illustrate the difference between applying the purge procedure just at the recipient or both at the recipient and at the sender. Figure 5.8 shows simulation results for a scenario where both the sender and the recipients have a buffer size of $N = 10$ and purging is performed at both ends and compares this with both a single buffer of $N = 10$ and $N = 20$. Notice that, since congestion propagates back from the bottleneck, purging is first performed exclusively at the receiver until the buffer fills up with un-

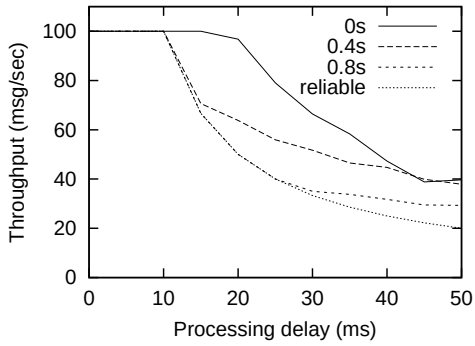


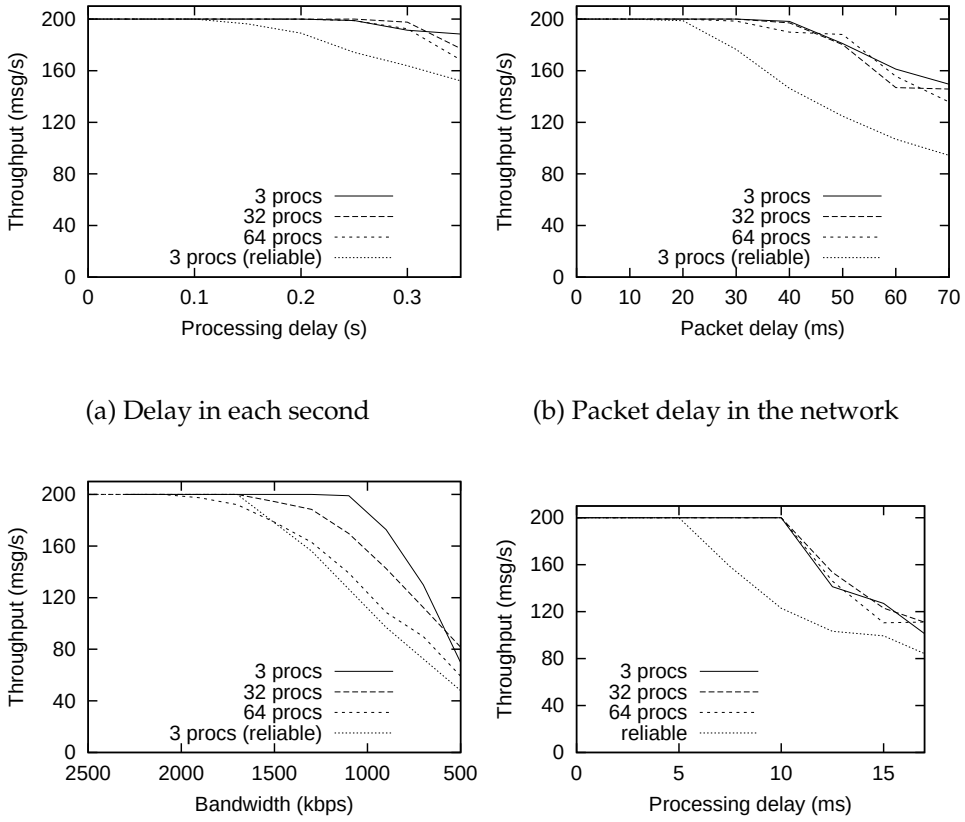
Figure 5.9: Impact of collecting majority of acknowledgments ($r = 0.5, d = 5$).

related messages. After that, back-pressure is exercised and messages start being purged also at the sender side. Although the result is better than a single buffer, it is not comparable with a buffer with twice the size. Nevertheless, it is important to allow purging in both buffers as it allows the system to cope also with network congestion.

Safety detection The requirement of determining safety prior to applying purging in retransmission buffers, that is implicit in FIFO Completeness, also affects purging, as messages cannot be used for purging immediately as they enter the buffer. The resulting effect is similar to a reduced buffer size, varying with the relation between safety latency and buffer latency. Figure 5.9 shows how throughput is reduced with an increasing safety detection time. Notice that with high safety detection latency, purging is effective only when the system is highly congested, as purging can be done only after messages have been stored for a long time.

5.4.2 Resource usage and scalability

We now focus on determining the issue of scalability of protocol mechanisms. We also address the impact of semantic reliability in the usage of resources such as processor and network. Both these issues are fundamental in estab-



(a) Delay in each second

(b) Packet delay in the network

(c) Network link bandwidth

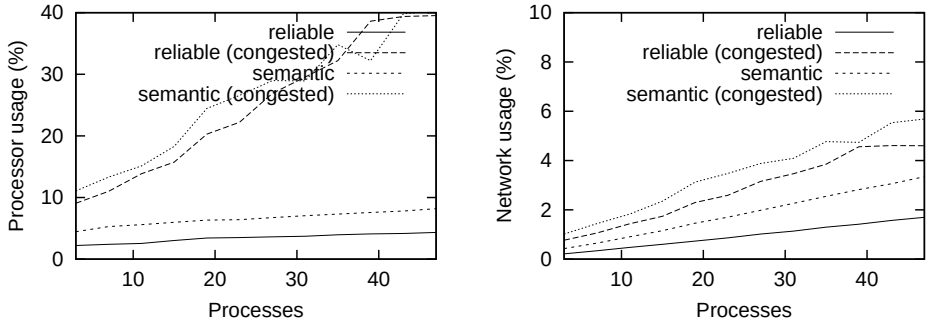
(d) Processing delay upon delivery

Figure 5.10: Impact of group size in throughput with various perturbations ($r = 0.5, d = 5$).

lishing that introducing semantic reliability does not otherwise compromise the performance of protocols.

Using the centralized simulation model it is possible to observe the behavior of the prototype implementation with a larger number of processes. Figure 5.10 presents the impact of semantic purging in face of several perturbations and large groups and can be compared to Figure 2.6. Results with the reliable protocol and 3 processes are displayed again for easier reference.

Figure 5.11 shows processor and network usage of both protocols as mea-



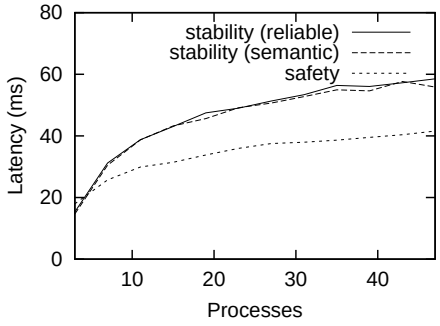
(a) Processor usage

(b) Network usage

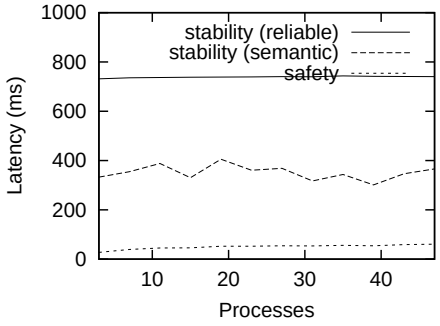
Figure 5.11: Impact of group size and purging in resource usage ($r = 0.5, d = 5$).

sured at the sender process. This has been obtained with small messages (4 bytes of payload) to highlight the overhead of protocol mechanisms. Processor usage includes time spent both in the protocol and in the operating system, but excludes the application. Most of the overhead of the reliable protocol when the system is not congested is attributable to the configuration chosen for stability tracking, which allows using very small buffers. Safety tracking, that is used only by semantic reliability, produces comparable overhead to stability tracking, thus doubling the protocol overhead when the system is not congested. The overhead can be lowered by decreasing the frequency of gossip rounds of the protocol and proportionally increasing buffer space to cope with increasing latency. In addition, if congestion is estimated from average buffer occupancy, safety tracking can be disabled when the system is not congested. When the system is congested, most of the overhead is attributable to retransmissions, which in this experiment are performed only by the sender, and thus the impact if safety tracking is not as relevant.

Figure 5.12(a) presents the average latency of stability and safety tracking when the system is not congested. As safety tracking latency is quite large



(a) Latency of stability and safety tracking



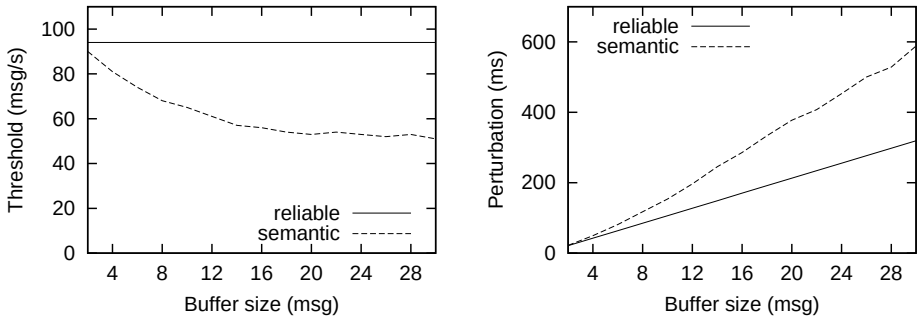
(b) Latency of stability and safety tracking when the system is congested

Figure 5.12: Impact of group size and purging in the latency of stability and safety tracking ($r = 0.5, d = 5$).

compared to stability tracking latency, purging of retransmission buffers is harder. This is not critical, as when the system is not congested purging is only used to accelerate garbage-collection. Figure 5.12(b) presents similar results with a single congested receiver. Stability tracking latency becomes dependent on the slower receiver. In this experiment safety tracking is not affected because there is a single slow receiver, thus purging opportunities can be fully exploited and greatly reduce stability latency in the semantic protocol. This is true as long as the number of fast receivers is greater than f.

5.4.3 View change frequency and latency

When the performance of a receiver is perturbed, applications using strict reliability have the option of expelling the perturbed member from the group to improve throughput stability [PS97]. This can be triggered by determining which are the processes that are delaying message stability. With S-VSM this can also be done, although it is not required as long as enough purging can be



(a) Threshold value

(b) Tolerated perturbation length

Figure 5.13: Impact of purging in the frequency of view changes.

done. It is thus interesting to consider, with the same traffic profile and system configuration, which is the size of a perturbation that triggers a view change.

The points of inflexion of the curves of Figure 5.3 indicate which is the minimal rate that a perturbed receiver has to ensure in order not to disturb the group and thus not to be expelled. If a receiver exceeds this threshold and remains in the group, its entire throughput is affected. For instance, with traffic with $r = 0.5$, this is 50 msg/s while for reliable it is 100 msg/s.

Figure 5.13(a) shows what is the lowest threshold value, for the degradation of a receiver, that can be tolerated (with less that 5% impact on the sender) as a function of the buffer size with traffic parameter $r = 0.5, d = 5$ when the input rate is 100 msg/s. In any case, with a reliable protocol, the receiver's rate can never be lower than the average input rate, otherwise it eventually slows down the system no matter how large the buffers are. On the other hand, with S-VSM, slower receivers can be accommodated by increasing the buffer size that enables purging to be done. Notice that S-VSM is not effective for very small buffer sizes due to the dependency on the distance among related messages.

The difference between the two lines of Figure 5.13(a) indicates the purg-

ing rate achieved by the protocol for each buffer size. The difference between the messages being produced and the messages being purged indicates the rate at which buffers fill-up for a given configuration. From this rate, we can also estimate the maximum length of the perturbation period that can be tolerated before the buffers are exhausted. As a function of the buffer size, Figure 5.13(b) shows for how long can be tolerated a receiver that completely stops to process messages. For instance, with a buffer size of 20 messages, a reliable protocol can only tolerate a perturbation of 212 ms while the S-VSM protocol can tolerate a perturbation of 377 ms.

The latency of view installation is related to the amount of used buffer space when a view change is triggered, as it must wait for all pending messages to be stable. If the view change is due to a slower process, then buffers are equally full and semantically reliability has no impact. Figure 5.7(b) presents the results of observing the amount of buffer used. Notice that for delays between 10 ms to 20 ms, when purging is enough to prevent throughput degradation, an eager strategy achieves this without buffers filling up.

5.5 Summary

We have analyzed the performance of our protocol using different approaches, namely using an analytical model, simulation and a concrete implementation. This experience allows us to draw conclusions about the validity of these approaches and identify the most relevant issues in the performance of semantically reliable protocols.

We have observed that the results from the prototype are sufficiently close to the simulation model to allow reliable estimations to be extracted from simulations. In addition, results show that the simple analytical model is useful in predicting the behavior of the system from the characteristics of the traffic. Therefore it can be used by application developers as a configuration tool.

This avoids having to use the more complex simulation model or configure the system by trial and error.

We have identified the following critical parameters in the performance of the protocol that have to be adjusted to the characteristics of the traffic: buffer size and maximum representable obsolescence distance. Configuration of the system can thus be done in two steps: *i)* a description of the traffic as a probability mass function of the distance between related messages is determined, either analytically or by profiling the application; *ii)* if probability of finding related pairs of messages accumulates in low values of distance, buffer size N and maximum obsolescence distance k can be selected as the minimum value which enables a sufficiently large share of related message pairs to be found.

We conclude that S-VSM allows longer perturbations to be tolerated with the same amount of allocated buffer space. Since this is achieved at the cost of purging obsolete information, and not at the cost of storing additional messages, S-VSM as no negative impact on the latency of the view change protocol.

Chapter 6

Case Study

The purpose of this chapter is to illustrate a concrete application of semantic reliability. This involves profiling the application to determine the obsolescence relation and then using it to evaluate the performance of the prototype. The case-study chosen is the replication of the server of a multi-player game.

6.1 Multi-player games

Multi-player games are an interesting application scenario where stringent performance and consistency requirements meet. These applications are not typically supported by group communication services, due to the following factors:

- Off-the-shelf group communication services have traditionally been targeted at applications without the stringent throughput requirements of highly interactive applications.

- High availability of servers has not been high in the list of priorities of game developers. In the past games were normally short-lived and servers managed on a best-effort basis, frequently by players themselves.

However, this scenario is bound to change as the number of multi-player games hosted by commercial services as well as the number of players and spectators in each game is growing. Therefore it would be convenient to use standard protocols for dissemination of game information. Also as a result of this trend, long lived games have been appearing in an attempt to keep players loyal to a server. In such systems, the need to preserve the server state and offer continuous service becomes an important concern. Therefore, it is extremely relevant to find abstractions that ease the task of replicating this type of servers in an efficient manner.

6.2 Replicated server

We consider a primary-backup approach to replicate the game server. To determine how the state is updated, we have inspected the code of the open-source game Quake™[1] to extract concrete obsolescence relations. The state of the game is modeled as a set of items. An item is any object in the game with which players can interact. The background is described separately and is immutable. Each item is represented by a data structure that stores its current position and velocity in the 3D space. The same data structure may also hold additional type specific attributes, such as the players remaining strength.

The game advances in rounds that correspond to frames that are displayed in players screens. Although the server tries to calculate 30 frames each second, this number can be reduced without loss of correctness. However, this degrades the perceived performance of the game hence the need to sustain a stable throughput. During each round the server gathers input from clients and re-calculates the state of the game. In each round, besides being updated, items can also be created and destroyed. For instance, when a bullet is fired an item has to be created to represent it, and if a player is later hit, both the

[1] `http://www.quake.com`

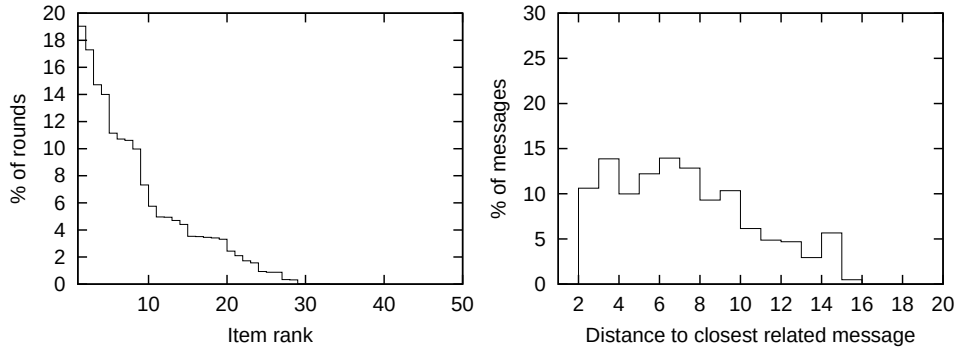
```
    /* Server process */

1:  upon reliable(data):
2:      seq ← seq+1
3:      multicast(REL(data)) tagged with (seq, seq, [0..0])

    /* Observer processes */

4:  upon deliver(REL(data)):
5:      ...
```

Figure 6.1: Addendum to pseudo-code of Figure 3.8.

items of the bullet and the player need to be removed. The transmission of the updated state includes:

- Updated values of items, for instance, as their position is altered. These make previous values of updates obsolete as they convey newer values.
- Destruction and creation of items. These must be reliably delivered to ensure that items are kept consistent.

Maximizing the amount of messages that become obsolete requires that each item is multicast as a separate message. This leads to bursty application traffic: In every round, after the new state is calculated, each modified item is multicast along with reliable messages. Notice that consistency requires that each round is applied atomically. Therefore we apply the scenario described in Section 2.3.2, representing the obsolescence relation as described in Section 3.4.3 that can easily be adapted to allow for some messages which never become obsolete as shown in Figure 6.1 through the addition of procedure *reliable(data)*.



(a) Frequency of item modifications (b) Obsolescence distance

Figure 6.2: Characterization of access to application state.

6.3 Traffic characterization

We have instrumented the game server to obtain experimentally the obsolescence patterns from real game sessions. These patterns are logged and later replayed, being only then multicast to a set of replicas. This is preferable to directly multicasting state updates from a live game, as it allows the same run to be repeated with different protocols and the results compared. By comparing the resulting state after applying messages delivered by S-RM with the state resulting from reliable multicast it is also possible to verify the correctness of the protocol implementation.

We detect which items are changed at each round by monitoring internal functions used to update the system state and to disseminate changes to clients. The results presented have been observed during a session with 5 players lasting for approximately 6 minutes and allowing us to record a total of 11696 rounds. This particular run was selected due to its length with a constant number of players.

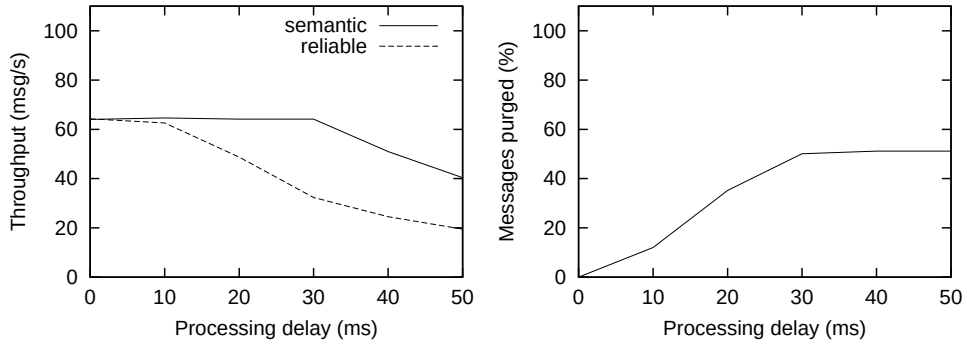
From the traffic generated it was observed that a share of 41.88% of the messages never became obsolete. The obsolescence pattern of the remaining

messages is related to the distribution of item updates. Although an average of 42.33 items were recorded active in each round, only an average of 1.39 items were modified. In addition, the results of Figure 6.2(a) show that a small number of items was modified frequently, while some items have not been modified at all during the measurement period. Therefore, consecutive updates of the same item are likely to be found close in the message stream. This is confirmed by Figure 6.2(b), that shows the distribution of distance between related messages. Notice that related pairs are usually close together (often within 10 messages of each other). All items are represented by the same data structure, which is 52 bytes long. This is the size of the message payload. Reliable messages have variable size depending on what happened in the round. Most were less that 32 bytes in size.

We have also collected data with other numbers of players. It can be observed that when more players join the game the message rate increases, the share of messages that never become obsolete decreases, but the distance between related messages increases. This suggests that higher purging rates would be possible that those presented here, although at the expense of larger buffer sizes.

6.4 Performance

Using the prototype implementation it is possible to observe the impact of purging in throughput stability. Figure 6.3(a) shows that semantic reliability allows the processing delay at the receiver to grow from 10 ms to 30 ms and to be accommodated without impacting the sender. This is explained by being able to purge 51.14% of traffic as shown in Figure 6.3(b). No purging is observed by fast receivers, because messages are rapidly delivered before a substituting message exists. Fast receivers are thus completely unaffected by congestion.

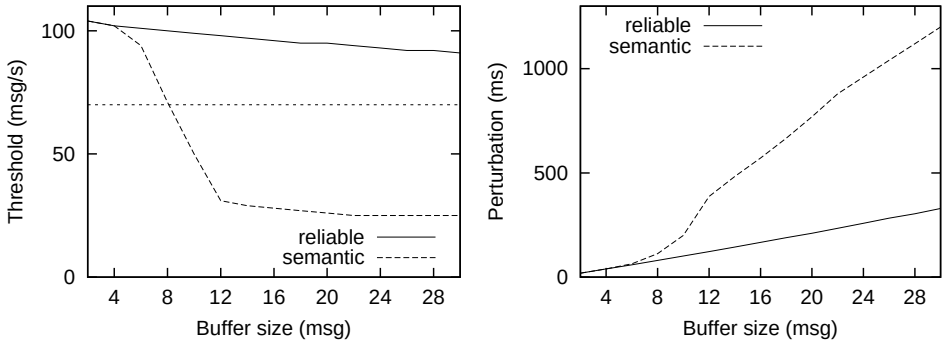


(a) Throughput (b) Purging

Figure 6.3: Performance of semantic reliability with Quake.

Of particular interest in these results are the points of inflexion of the curves of Figure 6.3(a). Figure 6.4(a) shows the lowest threshold value, for the degradation of a receiver, that can be tolerated (with less that 5% impact on the sender) as a function of the buffer size. The horizontal line shows the average rate of input traffic. In the presence of periodic traffic, a receiver could process messages at the average rate without affecting the group throughput. However, as it can be seen from the figure, due to the bursty nature of the game traffic pattern, when a reliable protocol is used, the receiver has to process messages at a faster pace (to accommodate the excess of messages during the bursts). As expected, it can be observed that larger buffers allow the reliable protocol to better accommodate message bursts. With semantic reliability, slower receivers can be accommodated by increasing the buffer size that enables purging to be done. Notice that semantic reliability is not effective for very small buffer sizes due to the distance among related messages.

As a function of the buffer size, Figure 6.4(b) shows for how long a receiver that completely stops to process messages can be tolerated. For instance, with a buffer size of 24 messages, a reliable protocol can only tolerate a perturbation of 342 ms while the S-VSM protocol can tolerate a perturbation of 857 ms. This



(a) Threshold value (b) Tolerated perturbation length

Figure 6.4: Impact of purging in the performance of view changes.

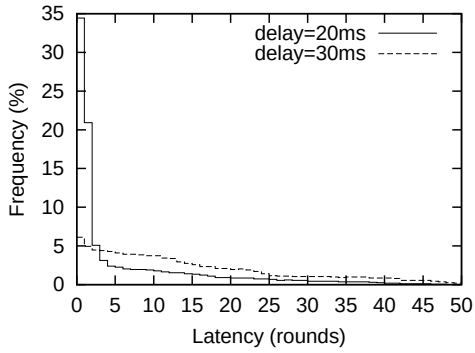


Figure 6.5: Latency histograms.

confirms that S-VSM allows longer perturbations to be tolerated with the same amount of allocated buffer space.

Although the application ends up receiving a message that obsoletes every purged message, this information is received with some additional delay. When semantic reliability is used to disseminate the information to clients, slower receivers will receive updates less often. Although this has no impact in the replication scenario outlined, we have measured it by counting the number of game rounds for which an item is outdated until the substituting message is received. This metric avoids the need for synchronized clocks.

When no purging exists, all modified items are updated within the round of modification. When the receiver is perturbed with 10 ms, 78% of modified items are updated within the a single round and 94% within 10 rounds. Results when purging is higher are presented in Figure 6.5.

6.5 Summary

This chapter presents the usage of semantic reliability in a concrete application, namely, replicating a server of multi-player game using the primary-backup approach. This involves determining and representing the obsolescence relation.

This application is a good example of a case where the reliability constraints conflict with other system requirements (in this case timeliness) leading, in the worst case, to a complete denial of service during load peaks. The notion of message obsolescence may provide the means to achieve a reasonable trade-off in this setting. Instead of introducing an arbitrary loss of messages, that could lead to losing information about some entities, obsolescence allows to introduce a selective purging of messages during congestion periods.

This is confirmed by performance results. Namely, it is possible to configure the system such that a server with approximately one third of the throughput can be accommodated without disturbing the performance of the group. With a buffer size of 24 messages, this translates to being able to to accommodate a completely disconnected member for 857 ms instead of 342 ms until the whole group is blocked.

Chapter 7

Conclusions

Reliable multicast protocols are useful in programming a wide range of distributed applications. Namely, in developing fault tolerant services by replication using a view synchronous multicast protocol as available in group communication toolkits. The deployment of reliable multicast is however challenged by environments with heterogeneous performance: A single slow node or network link can degrade the performance of the whole group by means of flow-control. This is often referred to as a "crying baby" and is an obstacle to the performance of systems requiring stable high throughput.

Previous proposals to address this problem fall roughly into three categories: temporarily expelling the perturbed member from the group; using very large buffers; and avoiding the usage of a reliable multicast protocol. Neither of these is suitable for fault tolerant applications.

Our proposal is to use knowledge about message semantics to selectively relax reliability. This allows us to improve throughput stability while keeping the convenient programming model of group communication. The approach is motivated by the observation that when the system is congested, buffers in the path to the bottleneck are full and thus are likely to contain messages that have been produced in different points in time. In many applications, recent messages implicitly convey the content or overwrite the effect of previ-

ous messages, which thereby become obsolete prior to their delivery to slow processes.

If obsolete messages can be recognized within protocol buffers and then purged, the application is relieved from processing some of the outdated messages and resources are freed to process further messages. Therefore, a recipient that suffers a performance perturbation does not prevent messages from stabilizing and can then be accommodated within the group without disturbing the remaining members. Purging of obsolete messages is not observed by fast members, which quickly deliver messages before becoming obsolete. This means that only slow processes omit deliveries: They do not receive all the messages but they still receive enough messages to be allowed to remain in the group.

The introduction of semantically reliable group communication is done in two steps:

- A suite of semantically reliable protocols is defined based on application semantics captured as a binary relation on messages. This includes combining semantic reliability with order and view synchrony. Important issues in implementing semantically reliable protocols are identified and discussed.

- The evaluation of performance of semantically reliable protocols is done by using simple analytical and simulation models and then by implementing and testing a multicast protocol.

Semantic reliability is effective only if obsolete messages can be found, *i.e.*, our solution is only effective for some applications in which messages frequently make recent messages obsolete. However, it can be observed that high throughput applications such as distributed multi-player games allow high purging rates. This is illustrated in two different ways:

- Typical applications of reliable multicast are examined for obsolescence and the necessary modifications for usage with a semantically reliable

protocol are presented as pseudo-code. This qualitative analysis serves also as a guide to programming with semantic reliability.

- A concrete application is profiled to determine the amount of purging that can be accomplished in real executions. This provides a quantitative example of the usefulness of semantic reliability.

An interesting conclusion is that existing group communication toolkits can be easily modified to provide semantic reliability. As semantic reliability defaults to strict reliability, this allows applications to run unmodified. This opens up the possibility of, in a second step, improving the performance of the application by defining suitable obsolescence relations.

7.1 Future work

As the emphasis of this dissertation is on the protocols themselves, the presentation of application programs is done mostly as motivation and as a programmer's guide. A formal treatment of the correctness of primary-backup and replicated state machine approaches based on semantic reliability is thus desirable. Specifically, proofs that both are linearizable [HW90].

Although the prototype implementation addresses protocol mechanism issues that arise in semantic reliability and allows the evaluation of performance, it lacks the functionality for being deployed, most notably, the absence of a view synchrony and total order. Nonetheless, the implementation of the complete semantically reliable group communication suite is underway, based on the Appia [MPR01] protocol composition framework.

Alternatively, the current prototype can be improved by using gossip instead of IP multicast for the first phase. This would allow it to simultaneous geographical and numerical large scale. The application of semantic purging to gossip-based multicast protocols offering probabilistic reliability guarantees is already being considered [PROK01, PROK02], allowing the reduction

of the bandwidth required for correct operation.

The concept of semantic reliability introduced in this thesis seems also to be generally useful in multicast protocols, even outside group communication. Namely, it is being considered to improve error-handling in wireless networks [EP01a, EP01b, EP02].

Bibliography

[AC97] G. Alvarez and F. Cristian. Applying simulation to the design and performance evaluation of fault-tolerant systems. In *IEEE International Symposium on Reliable Distributed Systems*, 1997.

[ADAD01] R. Arpaci-Dusseau and A. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Workshop on Hot Topics in Operating Systems*, May 2001.

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *IEEE International Symposium on Fault-Tolerant Computing*, July 1992.

[ADS00] Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *IEEE International Conference on Dependable Systems and Networks*, June 2000.

[AMMS+95] Y. Amir, L Moser, P Melliar-Smith, D. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4), November 1995.

[BCBT96] A. Basu, B. Charron-Bost, and S. Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *International Workshop on Distributed Algorithms*, October 1996.

[BDM01] O. Babaoglu, R. Davoli, and A. Montresor. Group communication in partitionable systems: Specification and algorithms. *IEEE Transactions on Software Engineering*, 27(4), 2001.

[BHO+99] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2), 1999.

[Bir93] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12), December 1993.

[Bir99] K. Birman. A review of experiences with reliable multicast. *Software Practice and Experience*, 29(9), July 1999.

[BMST93] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems*, chapter 8. Addison Wesley, 1993.

[BPRS98] R. Baldoni, R. Prakash, M. Raynal, and M. Singhal. Efficient Δ-causal broadcasting. *International Journal of Computer Systems Science and Engineering*, 13(5), September 1998.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3), August 1991.

[BvR94] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[CBDS01] B. Charron-Bost, X. Défago, and A. Schiper. Time vs. space in fault-tolerant distributed systems. In *IEEE International Workshop on Object-oriented Real-time Dependable Systems*, January 2001.

[CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, June 1999.

[CHT96] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4), July 1996.

[CKV01] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4), December 2001.

[CL85] K. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1), February 1985.

[Cla82] D. Clark. RFC 813: Window and acknowledgement strategy in TCP. IETF Request for Comments, July 1982.

[Cri91] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2), February 1991.

[CRW00] A. Carzaniga, D. Rosenblum, and A. Wolf. Content-based addressing and routing: A general model and its application. Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, January 2000.

[CT90] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, September 1990.

[CT96] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2), March 1996.

[DC90] S. Deering and D. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2), May 1990.

[Den68] P. Denning. The working set model for program behaviour. *Communications of the ACM*, 11(5), May 1968.

[EG02] P. Eugster and R. Guerraoui. Probabilistic multicast. In *IEEE International Conference on Dependable Systems and Networks*, June 2002.

[EGH+01] P. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kerrmarec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *IEEE International Conference on Dependable Systems and Networks*, June 2001.

[EMS95] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *IEEE International Conference on Distributed Computing Systems*, May 1995.

[EP01a] S. Elf and P. Parnes. A configurable transport layer as a cure for crying babies. Technical report, Luleå University of Technology, May 2001.

[EP01b] S. Elf and P. Parnes. A literature review of recent developments in reliable multicast error handling. Technical report, Luleå University of Technology, May 2001.

[EP02] S. Elf and P. Parnes. Applying semantic reliability concepts to multicast information messaging in wireless networks. In *IRMA International Conference*, May 2002.

[ES98] A. Erramilli and R. Singh. A reliable and efficient multicast for broadband broadcast networks. In *ACM Workshop on Frontiers in Computer Communications Technology*, August 1998.

[FB96] R. Friedman and K. Birman. Trading consistency for availability in distributed systems. Technical Report TR96-1579, Cornell University, Computer Science Department, April 1996.

[FJL+97] S. Floyd, Van Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6), December 1997.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 1985.

[FvR95] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in Horus. Technical Report TR95-1537, Cornell University, Computer Science Department, August 1995.

[GHvR+97] K. Guo, M. Hayden, R. van Renesse, W. Vogels, and K. Birman. GSGC: An efficient gossip-style garbage collection scheme for scalable reliable multicast. Technical Report TR97-1656, Cornell University, Computer Science Department, December 1997.

[GOS98] R. Guerraoui, R. Oliveira, and A. Schiper. Stubborn communication channels. Technical Report 98-278, Département d'Informatique, École Polytechnique Fédérale de Lausanne, 1998.

[GS95] R. Guerraoui and A. Schiper. Transaction model vs virtual synchrony model: bridging the gap. In K. Birman, F. Mattern, and

A. Schiper, editors, *Theory and Practice in Distributed Systems*, number 938 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

[GS97a] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4), April 1997.

[GS97b] R. Guerraoui and A. Schiper. Total order multicast to multiple groups. In *IEEE International Conference on Distributed Computing Systems*, May 1997.

[GS01] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1), January 2001.

[GT91] A. Gopal and S. Toueg. Inconsistency and contamination. In *ACM Symposium on Principles of Distributed Computing*, August 1991.

[Guo98] K. Guo. *Scalable Message Stability Detection Protocols*. PhD thesis, Cornell University, Computer Science Department, May 1998.

[Hay98] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, Computer Science Department, January 1998.

[HS95] M. Hiltunen and R. Schlichting. Properties of membership services. In *IEEE International Symposium on Autonomous Decentralized Systems*, April 1995.

[HT94] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.

[HvR95] T. Hickey and R. van Renesse. Incorporating system resource information into flow control. Technical Report TR95-1489, Cornell University, Computer Science Department, February 1995.

[HW90] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990.

[Jac88] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM Conference*, August 1988.

[Jai91] R. Jain. The art of computer systems performance analysis. *John Wiley & Sons, Inc.*, 1991.

[Lam78] L. Lamport. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM*, 21(7), 1978.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), May 1994.

[Lam97] L. Lamport. Processes are in the eye of the beholder. *Theoretical Computer Science*, 179(1–2), June 1997.

[LH99] K. Lin and V. Hadzilacos. Asynchronous group membership with oracles. In *International Symposium on Distributed Computing (DISC)*, 1999.

[Mal96] C. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks*. PhD thesis, Département d'Informatique, École Polytechnique Fédérale de Lausanne, 1996.

[MPR01] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *IEEE*

International Conference on Distributed Computing Systems, April 2001.

[NBT97] J. Nonnenmacher, E. Biersack, and D. Towsley. Parity-based loss recovery for reliable multicast transmission. In *ACM SIGCOMM Conference*, September 1997.

[Oli00] R. Oliveira. *Solving consensus: From fair-lossy channels to crash-recovery of processes*. PhD thesis, Département d'Informatique, École Polytechnique Fédérale de Lausanne, February 2000.

[OPS01] R. Oliveira, J. Pereira, and A. Schiper. Primary-backup replication: From a time-free protocol to a time-based implementation. In *IEEE International Symposium on Reliable Distributed Systems*, October 2001.

[Pat02] D. Patterson. Availability and maintainability >> performance: New focus for a new century. USENIX Conference on File and Storage Technologies, Keynote address, January 2002.

[Ped99] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, Département d'Informatique, École Polytechnique Fédérale de Lausanne, 1999.

[PGS02] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Journal of Distributed and Parallel Databases and Technology*, 2002.

[Pow96] D. Powell. Group communication. *Communications of the ACM*, 39(4), April 1996.

[PROK01] J. Pereira, L. Rodrigues, R. Oliveira, and A.-M. Kermarrec. Probabilistic semantically reliable multicast. In *IEEE International Symposium on Network Computing and Applications*, 2001.

[PROK02] J. Pereira, L. Rodrigues, R. Oliveira, and A.-M. Kermarrec. On the use of message semantics in probabilistic multicast. (Submitted for publication.), 2002.

[PS97] R. Piantoni and C. Stancescu. Implementing the Swiss Exchange Trading System. In *IEEE International Symposium on Fault-Tolerant Computing*, June 1997.

[PS99] F. Pedone and A. Schiper. Generic broadcast. In *International Symposium on Distributed Computing (DISC)*, September 1999.

[PTK94] S. Pingali, D. Towsley, and J. Kurose. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994.

[RBAR00] L. Rodrigues, R. Baldoni, E. Anceaume, and M. Raynal. Deadline-constrained causal order. In *IEEE International Symposium on Object-oriented Real-time distributed Computing*, March 2000.

[RGS98] L. Rodrigues, R. Guerraoui, and A. Schiper. Scalable atomic multicast. In *IEEE International Conference on Computer Communications and Networks*, October 1998.

[RM93] M. Raynal and M. Mizuno. How to find his way in the jungle of consistency criteria for distributed shared memories (or How to escape from Minos' Labyrinth). In *IEEE International Conference on Future Trends of Distributed Computing Systems*, September 1993.

[RM97] S. Raman and S. McCanne. Generalized data naming and scalable state announcements for reliable multicast. Technical Report CSD-97-951, University of California, Berkeley, June 1997.

[RSB93] A. Ricciardi, A. Schiper, and K. Birman. Understanding partitions and the "No partition" assumption. In *IEEE International Conference on Future Trends of Distributed Computing Systems*, September 1993.

[RV92] L. Rodrigues and P. Veríssimo. xAMp: a multi-primitive group communications service. In *IEEE International Symposium on Reliable Distributed Systems*, October 1992.

[Sat90] M. Satyanarayanan. A survey of distributed file systems. *Annual Reviews of Computer Science*, 4, 1990.

[SBS93] D. Schmidt, D. Box, and T. Suda. ADAPTIVE — A Dynamically Assembled Protocol Transformation, Integration and eValuation Environment. *Concurrency: Practice and Experience*, 5(4), 1993.

[Sch93] F. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*, chapter 7. Addison Wesley, 1993.

[SKM00] J. Sussman, I. Keidar, and K. Marzullo. Optimistic virtual synchrony. In *IEEE International Symposium on Reliable Distributed Systems*, October 2000.

[SS93a] A. Schiper and A. Sandoz. Understanding the power of the virtually-synchronous model. In *European Workshop on Dependable Computing*, February 1993.

[SS93b] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *IEEE International Conference on Distributed Computing Systems*, May 1993.

[TTP+95] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly con-

nected replicated storage system. In *ACM Symposium on Operating Systems Principles*, December 1995.

[WS95] U. Wilhelm and A. Schiper. A hierarchy of totally ordered multicasts. In *IEEE International Symposium on Reliable Distributed Systems*, September 1995.

Appendix A

Correctness Proofs

Liveness properties of S-RM are sufficiently different from those of reliable mulitcast to justify a detailed proof. We use the opportunity to relax the assumption of FIFO channels thus introducing additional detail in the algorithm. On the other hand, we resort to a more abstract notation to simplify the presentation of the proof. Appendix B discusses how a practical implementation can be obtained from this algorithm.

A.1 System model and notation

The system execution is modeled as a sequence of states [Lam94]. Each state is a mapping from state variables to values. A next state relation is a predicate on pairs of states. A specification is a set of executions, that can be defined by a next state relation which is true for consecutive states in legal executions, plus fairness assumptions written in temporal logic.

The state describes both the algorithm and the environment. Processes and channels are not explicit: A process state is a portion of the system state and channel operations are modeled as copying elements between the state of two processes [Lam97]. Process crash is denoted explicitly by state variables. A process is considered correct if its crashed state is forever false.

We use the common notation for sets. For tuples, we use the usual notation π_n to denote projection of element n. This notation is extended for sets of tuples with Π_n to denote the set of projections. For sequences, we use $\langle m \rangle$ to denote a sequence with one element m and $\circ$ to denote concatenation. $elems(S)$ denotes the set of elements of a sequence S. A message multicast by process i is expressed as $m \in elems(M_i)$ and a message delivered by process i is expressed as $m \in elems(D_i)$. Likewise, order of multicast and delivery are expressed as ordering in sequences M_i and D_i.

A.2 Algorithm

The challenges in implementing S-RM can be summarized by two example scenarios of how a naive implementation, that would just delete from the buffers messages made obsolete by the reception of a subsequent message, would fail.

The first is the following scenario and arises because the assumption of FIFO order in point-to-point channels is not made: *i)* a process p multicasts two unrelated message m_1 and m_2 ($m_1 \not\sqsubset m_2$); *ii)* the same process p multicasts an infinite sequence of messages $m_3, m_4, \ldots$ such that $m_1 \sqsubset m_3$ and $\forall i \geq 3 : m_i \sqsubset m_{i+1}$. Consider that p purges m_1 from its buffer before sending both m_1 and m_2 to another process q. Since m_2 was sent after m_1, q will not deliver m_2 before m_1 arrives (that would never occur) or until it realizes that m_1 was purged because it was made obsolete by some other message m_i. However, since m_i belongs to an infinite sequence, it may never arrive at q. Therefore, the protocol must incorporate some mechanism to ensure that q is informed about the purging of m_1.

Even if information about purged messages is propagated, it is possible to show that the naive implementation would not ensure both Semantic Validity and Semantic FIFO Completeness in the case of failures. Consider the same

scenario as above and the following sequence of events: *i)* process p multicasts m_1, m_2 and m_3; *ii)* p purges m_1 due to m_3 and informs the remaining processes that m_1 was purged; *iii)* p sends m_2 that is delivered by some process q; *iv)* p crashes. Clearly, this sequence violates Semantic FIFO Completeness. The problem is that m_1 was purged before ensuring the delivery of m_3. A message is guaranteed to be eventually delivered as soon as it has been received by $f + 1$ processes, where f is the maximum number of processes that may fail. When this condition holds, we say that the message is *safe*. In the particular sequence above, violation of Semantic FIFO Completeness could be avoided if purging of m_1 was delayed until m_3 was known to be safe.

Given these observations, our protocol is based on the following principles:

- As in any reliable protocol, processes forward all the messages they received to mask the failure of the sender.
- In a retransmission buffer, a message m may be purged only if there is another message m' such that: $m \sqsubset m'$ and m' is safe.
- When a message is purged, enough information is stored to inform the remaining processes that the message has been purged.

The variables used by each process i are listed in Figure A.1. Variable M_i simply keeps the messages that have been multicast by the process. The variable c_i records the state of the process (false if the process is correct and true if the process is crashed). Each process keeps a pair of buffers $I_{i,j}$ and $O_{i,j}$ for each process j. Variable $I_{i,j}$ is an incoming buffer from j, where messages waiting to be ordered are stored. Variable $O_{i,j}$ stores messages waiting to be transmitted to j. Messages ordered are copied to a local delivery queue Q_i and messages that have been delivered are recorded in D_i. Messages in $O_{i,j}$ and Q_i can be purged. Purging is modeled by changing an attribute that is associated with each message in a given queue. This attribute can have one of two

State for each process i:

M_i: messages multicast, initially empty sequence
D_i: messages delivered, initially empty sequence
c_i: crashed state, boolean, initially false
$O_{i,j}$: outgoing toward j, initially empty
$I_{i,j}$: incoming via j, initially empty
Q_i: queued for delivery, initially empty

Figure A.1: State variables.

TE1: $transmit_{j,i}(m, o)$
PRE-CONDITION:
$(m, o) \in O_{j,i} \wedge \neg c_j \wedge$
$m \notin \Pi_1(I_{i,j}) \wedge \neg c_i$
EFFECT:
$I_{i,j} := I_{i,j} \cup \{(m, o)\}$

TE2: $crash_i$
PRE-CONDITION:
$|\{j : c_j\} \cup \{i\}| < f$
EFFECT:
$c_i := true$

Figure A.2: Transitions associated with the environment.

values: D (the message contains data) or P (the message has been purged).

Figure A.2 depicts the transitions of the environment. Transition TE1 simply specifies that messages in output buffers are eventually inserted in the corresponding input buffers from the destination processes (this models the transmission of messages in the links). Transition TE2 specifies that a process may crash as long as the maximum number of faulty processes has not been reached.

Figure A.3 depicts the transitions for each process i:

- Transition TP1 corresponds to the multicast of a message m. In this transition the fact that m has been multicast is stored in M_i and the message is sent to self by inserting it in $O_{i,i}$ (note that the environment will eventually move the message to $I_{i,i}$). Notice that the predicate $next(m, S)$, that ensures that all predecessors of message m are available in set S, is used to enforce that the message being multicast has the right sequence number.

- Transition TP2 captures the forwarding procedure executed by every node. When a message is received for the first time and it is the next message in the sequence, as enforced by $next(m, S)$, it is copied to all output buffers and inserted in Q_i for delivery.

- Transition TP3 captures the delivery of messages (note that, in practice, when a purged message is delivered the application is not disturbed).

- Transition TP4 specifies that a message m, waiting to be delivered, can be purged as long as in the same queue there is a subsequent message m' that makes m obsolete.

- Finally, transition TP5 specifies that a message m in an output buffer can only be purged if in the same queue there is a subsequent message m' that makes m obsolete *and* m' is safe. This is ensured by predicate *safe*(m, i), which checks that process i has received m from more than f processes.

The fairness assumptions for the algorithm are the following. No fairness assumptions for $multicast_i(m)$, $purge\_q_i(m)$, $purge\_r_i(m)$ and $crash_i$, thus allowing them to be forever enabled but never executed. Weak fairness is assumed for $transmit_{j,i}(m, o)$, for all i, j, m, o, and for $enqueue_i(m, o)$/ $deliver_i(m)$, for all i, m, o. This requires them to be eventually executed if forever enabled. Notice that there is no fairness imposed on purging operations, thus allowing reliable executions where no message is discarded.

A.3 Proof

We focus on proving liveness properties of the specification because these are the ones that make the difference to strict reliability and are those that can be compromised by losing messages.

TP1: $multicast_i(m)$
PRE-CONDITION:
$next(m, O_{i,i}) \wedge \neg c_i$
EFFECT:
$M_i := \langle m \rangle \circ M_i$
$O_{i,i} := O_{i,i} \cup \{(m, \mathrm{D})\}$

TP2: $enqueue_i(m, o)$
PRE-CONDITION:
$\exists j : (m, o) \in I_{i,j} \wedge m \notin \Pi_1(Q_i) \wedge$
$next(m, Q_i) \wedge \neg c_i$
EFFECT:
for all $k \neq i$:$O_{i,k} := O_{i,k} \cup \{(m, o)\}$
$Q_i := Q_i \cup \{(m, o)\}$

TP3: $deliver_i(m)$
PRE-CONDITION:
$(m, \mathrm{D}) \in Q_i \wedge m \notin elems(D_i) \wedge \neg c_i$
EFFECT:
$D_i := \langle m \rangle \circ D_i$

TP4: $purge\_q_i(m)$
PRE-CONDITION:
$\exists m' : (m, \mathrm{D}), (m', \mathrm{D}) \in Q_i \wedge$
$m \sqsubset m' \wedge \neg c_i$
EFFECT:
$Q_i := (Q_i \setminus \{(m, \mathrm{D})\}) \cup \{(m, \mathrm{P})\}$

TP5: $purge\_r_i(m)$
PRE-CONDITION:
$\exists m' : (m, \mathrm{D}), (m', \mathrm{D}) \in O_{i,j} \wedge$
$m \sqsubset m' \wedge safe_i(m') \wedge \neg c_i$
EFFECT:
$O_{i,j} := (O_{i,j} \setminus \{(m, \mathrm{D})\}) \cup \{(m, \mathrm{P})\}$

safe$(m, i) =$
$|\{j : (m, o) \in I_{i,j}\}| > f$

$next(m, S) =$
$\forall s < seq(m), \exists m \in \Pi_1(S) :$
$snd(m) = snd(m') \wedge seq(m') = s$

Figure A.3: Transitions associated with process i.

Of the remaining properties, Integrity is trivially satisfied. The correctness of FIFO Order derives from *i)* Q_i containing a complete prefix of messages multicast and *ii)* a message that is purged in Q_i is never available as data (m, D) again in Q_i.

The proof of each of the liveness properties of the specification requires that if some condition on a message m is true, then some message m' such that $m \sqsubseteq m'$ is eventually delivered. This is split in two steps:

1. We prove that if for a pair of correct processes j, i, $m \in \Pi_1(O_{j,i})$ and there is a time after which no process multicasts m'' such that $m \sqsubset m''$, then eventually exists some $m' \in elems(D_i)$ such that $m \sqsubseteq m'$.

2. For each specification property, we prove that the condition it imposes on m implies that for some pair of correct processes j, i, $m \in \Pi_1(O_{j,i})$.

This makes the proof associated with the first step in Lemma 3 the only eventuality proof required. This proof uses the results of two auxiliary lemmata that summarize interesting aspects of the protocol. The proofs use some additional notation: The *path* to a process i, denoted H_i, is defined as $H_i = \bigcup_{\neg c_j}(O_{j,i} \cup I_{i,j})$. The *world* W is $\bigcup_{i=0,j=0}^{n,n}(I_{i,j} \cup O_{i,j})$. The predecessors of a message m are $Pred(m) = \{m' \in M : snd(m') = snd(m) \wedge seq(m') < seq(m)\}$.

Lemma 1 *If $(m, \mathrm{P}) \in H_i$ then there is some m' such that $m \sqsubset m'$ and for every process j (correct or not) $m' \in \Pi_1(H_j)$.*

PROOF: If $(m, \mathrm{P}) \in H_i$ then for some process k, $(m, \mathrm{P}) \in O_{k,i}$ or $(m, \mathrm{P}) \in I_{i,k}$. Moreover, if $(m, \mathrm{P}) \in I_{i,k}$ then $(m, \mathrm{P}) \in O_{k,i}$. This is true as *i)* (m, P) is never removed from $O_{k,i}$ and *ii)* the only action that inserts elements in $I_{i,k}$ is only enabled if the same element is in $O_{k,i}$.

Trivially if $(m, \mathrm{P}) \in O_{k,i}$ then $(m, \mathrm{P}) \in W$. If $(m, \mathrm{P}) \in W$ then there is some m', $m \sqsubset m'$, and a set of processes L with $|L| > f$, such that for any $l \in L$, $m' \in I_{k,l}$. This is true as *i)* the only action that inserts (m, P) in W is only enabled when m' is in more than f incoming queues and *ii)* if $m' \in \Pi_1(I_{k,l})$ once, then it is forever true. Therefore, for any process $l \in L$, $m' \in O_{l,k}$ and there is at least one $l \in L$ that is correct, as crash is enabled only for f processes.

If $m' \in \Pi_1(O_{l,k})$ then for all j, $m \in O_{l,j}$. This is true as *i)* transition $enqueue_j(m, o)$ always inserts m in all $\Pi_1(O_{l,j})$. Therefore, for any j, $m' \in \Pi_1(H_j)$. $\square$

Lemma 2 *Any path H_i contains a complete sequential prefix of the message ordering: For all $m \in \Pi_1(H_i)$, $Pred(m) \subset \Pi_1(H_i)$.*

PROOF: For all i, $\Pi_1(Q_i)$ is a prefix of the ordering. This is true as the only action that changes it is only enabled when the new message is the next in the sequence. Moreover, $\Pi_1(O_{j,i} \cup I_{i,j})$ is always equal to $\Pi_1(Q_j)$. The only actions that change $\Pi_1(O_{j,i} \cup I_{i,j})$ also change $\Pi_1(Q_j)$ accordingly. Therefore,

as the union of prefixes is still a valid sequential prefix, any path H_i contains a prefix. $\square$

Lemma 3 *If forever* $m \in \Pi_1(H_i)$ *then eventually* $m' \in elems(D_i)$ *such that* $m \sqsubseteq m'$.

PROOF: We define a set of tuples $Stat(m) \subseteq P \times M \times 2^M \times 2^M \times 2^M \times 2^M$ such that $(r, x, s_0, s_1, s_2) \in Stat(m)$ iff $m \sqsubseteq x$; $\bigcup_{i=0}^{3} s_i = Pred(x) \cup \{x\}$ and $\forall i \neq j : s_i \cap s_j = \emptyset$.

We define a relation $\prec$ in $Stat(m)$ such that $t \prec t'$ iff either $\pi_1(t) \subset \pi_1(t')$; or $\pi_1(t) = \pi_1(t')$ and $\pi_1(t) \sqsubset \pi_1(t')$; or $\pi_1(t) = \pi_1(t')$ and $\pi_1(t) \not\sqsubset \pi_1(t')$ and for some a for all $3 \leq a < b \leq 5$, $\pi_a(t) = \pi_b(t')$ and $\pi_a(t) \subset \pi_b(t')$. If the set of messages that make m obsolete is finite, then $Stat(m)$ is also finite. $(Stat(m), \prec)$ is a strict partial order because both strict set inclusion and obsolescence are strict partial orders. Thus $(Stat(m), \prec)$ is well-founded.

We now define a function $f_{i,m}$ from system state to $Stat(m)$ defined for states in which process i has not crashed and $m \in \Pi_1(H_i)$. Let $f_{i,m} = (r, x, s_0, s_1, s_2)$ such that:

- $r = \{i : \neg c_i\}$
- choose $x \in \Pi_1(H_i)$ such that $m \sqsubseteq x$ and $\forall m' \in \Pi_1(H_i) : x \not\sqsubset m'$;
- $s_0 = \Pi_1(\bigcup_{k \in c} O_{k,i}) \setminus \Pi_1(\bigcup_{k \in c} I_{i,k}) \cap Pred(x)$
- $s_1 = \Pi_1(\bigcup_{k \in c} I_{i,k}) \setminus \Pi_1(Q_i) \cap Pred(x)$
- $s_2 = \Pi_1(Q_i) \setminus D_i \cap Pred(x)$

Assuming that $m \in \Pi_1(O_{j,i})$ such that j is correct (forever $\neg c_j$), we prove that eventually $m' \in elems(D_i)$ by ensuring that *i)* if $m \in O_{j,i}$ then $f_{i,m} \in Stat(m)$, that is true by definition; *ii)* for some helpful transitions either $f_{i,m} \prec f'_{i,m}$ or $m' \in elems(D_i)$ and at least one is enabled or $m' \in D_i$; and *iii)* for the remaining transitions, never $f'_{i,m} \prec f_{i,m}$.

The transitions considered helpful and respective resulting values for $f_{i,m} = (r, x, s_0, s_1, s_2)$ are:

- $transmit_{j,i}(m', o)$ if $m' \in s_0$, leads to $(r, x, s_0 \setminus \{m'\}, s_1 \cup \{m'\}, s_2)$.
- $enqueue_i(m', o)$ if $m' \in s_1$, leads to $(r, x, s_0, s_1 \setminus \{m'\}, s_2 \cup \{m'\})$.
- $deliver_i(m')$ if $m \not\sqsubseteq m' \wedge m' \in s_2$, leads to $(r, x, s_0, s_1, s_2 \setminus \{m'\})$. Notice that $m' \not\sqsubseteq m$ implies $m' \neq x$.
- $deliver_i(m')$ if $m \sqsubseteq m'$. Goal reached with $m' \in elems(D_i)$.

At least one of these is enabled: If $transmit_{j,i}(m, o)$ is not enabled for all j, then at least $m \in \Pi_1(I_{i,j})$ for some j as $m \in \Pi_1(H_i)$. If $enqueue_i(m, o)$ is also not enabled, then $m \in \Pi_1(Q_i)$. Otherwise, with $m \in \Pi_1(H_i)$ and H_i containing complete prefixes (by Lemma 2), transmission would have to be enabled. If $delivery_i(m), m \not\sqsubseteq m'$ is also not enabled then $s_2 = \{x\}$, as Q_i contains x. Otherwise $delivery_i(x)$ is enabled as x must be tagged with D. Otherwise (by Lemma 2) it would not be a maximal element.

There are transitions that help but are not guaranteed to occur. Either because there is no fairness, namely in $multicast_i(m')$ if $x \sqsubset m'$ and $crash_j$ if $\neg c_j$, or are fair but may never be enabled, namely $enqueue_k(m')$ if $x \sqsubset m'$. Other actions leave $f_{i,m}$ unchanged. Notice that $crash_i$ does not happen by assumption. □

Theorem 1 (Semantic Validity) *If a correct process multicasts a message m and there is a time after which no process multicasts m'' such that $m \sqsubset m''$, then eventually it delivers some m' such that $m \sqsubseteq m'$.*

PROOF: It is trivially true that if $m \in M_i$ then $m \in O_{i,i}$ and process i is correct by assumption. Proof follows immediately from Lemma 3. □

Theorem 2 (Semantic Agreement) *If a correct process delivers a message m and there is a time after which no process multicasts m'' such that $m \sqsubset m''$, then all correct processes eventually deliver some m' such that $m \sqsubseteq m'$.*

PROOF: By a simple invariance proof, if i delivers m then $m \in O_{i,j}$ for all j and process i is correct by assumption. Proof follows immediately by Lemma 3. □

Theorem 3 (Semantic FIFO Completeness) *If a process multicasts a message m before it multicasts a message m' and there is a time after which no process multicasts m''' such that $m \sqsubset m'''$, no correct process delivers m' without eventually delivering some m'' such that $m \sqsubseteq m''$.*

PROOF: By a simple invariance proof (same as Semantic Agreement), if i delivers m' then $m' \in O_{i,j}$ for all j. By Lemma 2, the same $O_{i,j}$ contains m. Thus either m is delivered or by Lemma 1 some m'' such that $m \sqsubseteq m''$ exists. □

A.4 Causal order

The algorithm used for S-RM with FIFO order can easily be adapted for causal order. In detail, it is necessary to be able to determine the set of predecessors *Pred*(m) of a given message. This reduces to knowing for each message m the sequence number of the last message from each process j delivered to *snd*(m) at the time that m is multicast. Let this be $cseq(m, j)$. Then $Pred(m) = \{m' \in M : (snd(m') = snd(m) \wedge seq(m') < seq(m)) \vee (\exists j \in P : snd(m') = j \wedge seq(m') \leq cseq(m, j))\}$. The changes to the algorithm are then reduced to the definition of predicate *next*(m, S) to reflect the redefinition of predecessor set. The impact of the redefinition of $Pred(m)$ is restricted to Lemma 2 and the result remains valid, as all predecessors of a message are necessarily contained in the same path.

Appendix B

Implementation Details

The abstract presentation of the algorithm makes several simplifications, such as assuming that information about past messages indefinitely accumulates in the variables at each process. The specification also requires that information about purged messages is always explicitly sent over the network. We now argue how a practical implementation can be derived from the specification. For clarity, we address first the case where no purging occurs before discussing how purging can be implemented.

B.1 Window-based implementation

In the algorithm of Figures A.2 and A.3, sets $O_{j,i}$, $I_{i,j}$ and Q_i represent a point-to-point FIFO reliable channel as follows: Messages currently in transit (sent but not yet received), are $O_{j,i} \setminus Q_i$. Messages available only at the sender side are $O_{j,i} \setminus I_{i,j}$. Messages available at the receiver side waiting to be ordered are $I_{i,j} \setminus Q_i$. Notice that operations $transmit$ and $enqueue$ never refer to the content of messages in Q_i but need only the knowledge of the sequence number of the last message delivered.

In practice, this can be implemented using a pair of buffers (one on each side of the channel) and a sequence counter on the receiver: *i)* messages sent

are placed in the outgoing buffer, being eventually sent and if necessary repeatedly resent to the network (this is first line of $enqueue$); *ii)* upon reception, an acknowledgment is sent back and if necessary, repeatedly resent; *iii)* upon reception of acknowledgment the message is removed from the sender buffer (this implements $transmit$); *iv)* when a message bearing the next sequence number is available at the receiver, it is removed from the buffer and the sequence number is incremented (this implements the second line of $enqueue$).

Therefore, it is possible to implement the abstract specification of a channel using a window-based protocol. Since, in the proposed algorithm, there is symmetric connection (*i.e.*, $O_{i,j}$, $I_{j,i}$ and Q_j), acknowledgments can be piggybacked on messages traveling in the opposite direction as happens in TCP/IP in which acknowledgments are implicit in the lower bound of the window.

B.2 Purging

When purging happens in $O_{j,i}$, (m, D) is replaced by (m, P). In practice, for purging to be useful this must be implemented as freeing all resources (memory and bandwidth) consumed by m. This can be done in the sender's buffer, thus preventing network resources from being wasted. However, the receiver has to be notified that m has been purged to advance the sequence counter without receiving m.

This can be done using the following strategy: *i)* assume a fixed window of size w: the sender never puts sequence $s+w$ in the network without previously receiving an acknowledgment to s; *ii)* the sender knows that it has not received an acknowledgment s if some m such that $seq(m) = s$ is in the buffer; *iii)* if m is purged, it is removed from the buffer thus allowing $s + w$ to be put in the network and eventually received. When the receiver gets $s + w$ without ever getting s it must conclude that the message with sequence s has been purged. This implements (m, P) being inserted in $I_{i,j}$. Notice that no message

labeled with P is, in the algorithm, ever used for anything besides inspecting its sequence number, that in practice translates to it not occupying space. Note also that if there are no further messages to send, a message indicating that the window is empty needs to be explicitly sent.

Likewise, Q_i and D_i abstract a FIFO queue holding messages $Q_i \setminus D_i$ ordered by sequence number. Messages are inserted by $enqueue$ and removed by $deliver$. Purging in this buffer is implemented by removing the purged message.

B.3 Multicast network

If multicast links are available, it is also possible to optimize the message forwarding procedure that, in the abstract specification, requires each message to be transmitted on the network n^2 times. Two optimizations are possible:

- As a message is always simultaneously inserted in all outgoing channels $O_{i,j}$, a network level multicast mechanism can be used to transmit it, thus reducing the complexity to n.
- As soon as a message is received in some $I_{i,j}$ it can be acknowledged in all incoming channels: the receiver advances the lower bound of the window thus allowing the sender to immediately remove the message from its buffers. This also reduces the complexity to n.

Using both optimizations simultaneously, a message can be transmitted only once in the network. In addition, as in conventional reliable multicast protocols, the explicit point-to-point acknowledgment mechanism can be replaced by a global stability tracking mechanism thus further improving performance and scalability.