Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XIX Ciclo – 2006

Protocols for End-to-End Reliability in Multi-Tier Systems

Paolo Romano

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XIX Ciclo - 2006

Paolo Romano

Protocols for End-to-End Reliability in Multi-Tier Systems

Thesis Committee

Prof. Francesco Quaglia (Advisor)
Prof. Bruno Ciciani
Prof. Silvio Salza

Reviewers

Prof. Dhiraj K. Pradhan
Prof. Mukesh Singhal

Author's address:
Paolo Romano
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
e-mail: paolo.romano@dis.uniroma1.it
www: http://www.dis.uniroma1.it/∼romanop

To Francesco and Iolanda.

Acknowledgments

My deepest debt of gratitude goes to Francesco Quaglia for being a great advisor. His precious guidance, continuos support and enlightening wits had a major influence on this thesis: not only Francesco taught me how to approach research problems, but also provided me with constant encouragement to overcome the difficulties which arose throughout my doctoral program.

Words fail to express my gratitude to Bruno Ciciani. His invaluable technical and fatherly advices always accompanied me during the whole Ph.D. course and helped me developing both as a person and as a computer scientist.

I am also grateful to Dhiraj Pradhan and Mukesh Singhal for having accepted to serve as external referees of this dissertation, and to Silvio Salza for having served as third member of my thesis committee.

Many thanks to Milton Romero, Valeria Cardellini and Michele Colajanni. During my Master thesis, in early times of doubt and confusion, their ardor for research and academic life had a strong influence in persuading me to pursue my Ph.D.

My gratitude goes also to Suresha and Jayant Haritsa for providing me with the source code of their real-time database simulator, and to Mark Allman, Jay Aikat and Sushant Rewaskar for granting me access to the raw traces of empirical RTT measurements obtained in their previous works. Their contributions served as a basis to develop the simulation models employed in Chapter 5.

I cannot forget the many roommates whose pleasant company I could enjoy during these years, while roaming across three different rooms of our department. In particular I wish to thank Fabio "Tsukahara" Patrizi and Andrea "Raistlin" Santoro, for their warm friendliness during this last year: their jokes, laughters and constant support greatly alleviated the stress experienced while preparing this thesis.

A thought and a praise go to Wind and to Waves: beloved companions of adrenalinic (wind)surfing escapes in search of the marvel and of the fury

of elements, coveted prizes for successful endeavors and sweet soothers of less fortunate undertakings.

Herein I wish to take a moment to remember and deeply thank Marco Cadoli for his exemplary courage and strength, his uncommon amiability and humilty, his extraordinary commitment to academic activities even in the roughest times. His teachings, his memory will always live in our hearts.

Anna deserves a warm and special acknowledgment for her closeness during good and bad moments throughout all these years. Her smiles, her energy and her confidence in my abilities have always instilled in me the strength to face adversities. Not to mention her valuable help with many of the graphical elements appearing in this thesis, and her endless patience in bearing with me during the many days and nights of hard work chasing impossible deadlines.

Frankly there wouldn't have been any thesis of mine without the encouragement, support and love of my parents, Francesco and Iolanda, and of my brother, Elio. I wish I knew some way of returning even a fraction of what they have given me.

Rome, Italy Paolo
December 14th, 2006

Abstract

Modern Internet services exhibit the strong trend to be structured according to a three-tier, and in general multi-tier, system organization, which allows reflecting at both the software and hardware level the logical decomposition of applications. Even though the partitioning of the application into multiple tiers provides the potentialities to achieve high modularity and flexibility, the multiplicity and diversity of the employed components, and their interdependencies, make reliability a complex issue to tackle. As an example, in classical client-server environments, database systems represented the reliability backbone of mission critical services, ensuring consistent evolution of the state trajectory of business applications through the notion of atomic transactions. However, the fault-tolerance capabilities provided by transactional components, and, in broader sense, by traditional approaches to reliability, address issues restricted to specific subsystems involved in the end-to-end interaction. Hence, they are unable to tackle the wide spectrum of failure scenarios that can arise along the whole chain of components constituting a multi-tier system.

The design of reliability solutions for Internet services is made even more challenging by the open, heterogeneous and inherently asynchronous nature of the Internet itself, which dramatically reduces the possibility to monitor and control the distributed components involved in a multi-tier application. Further, coupled with global access enabled by the Internet and with widespread diffusion of complex services, the urge for achieving high scalability and minimizing response times has accordingly grown. This has imposed stringent performance requirements on the underlying reliability mechanisms.

This is precisely the focus of this thesis. Specifically, we introduce innovative protocols ensuring the e-Transaction (exactly-once Transaction) guarantees, namely a recent formalization of desirable end-to-end reliability properties for multi-tier systems in presence of crash failures. These protocols advance the state of the art in a twofold direction. From a practical perspective, they achieve unparalleled scalability levels, exhibit very limited overhead, thus revealing particularly attractive in the context of emerging large scale service delivery platforms. From a theoretical standpoint, our solutions can cope with purely asynchronous systems, where no assumption on the accuracy of

the failure detection mechanism can be guaranteed.

As we will show, some of the building blocks underlying the previous fault tolerant protocols can also be used to construct distributed protocols allowing the treatment of a more general class of failures, which we can refer to as "performance failures". These model situations of reduced system responsiveness due to both crashes and overloads/congestions on some component.

The remainder of this document is structured as follows. In Chapter 1 and Chapter 2 we discuss (end-to-end) reliability issues in multi-tier systems, and provide an overview of the related state of the art approaches.

Original contributions of this document start from the following chapter. Specifically, in Chapter 3 and Chapter 4 we present two innovative e-Transaction protocols coping with the general case of multiple, autonomous databases in the system back-end. Trade-offs between these two solutions are discussed for what concerns overhead, integration with conventional technology, and requirements on the synchrony level of the underlying system.

In Chapter 5, we show how the building blocks provided in the previous chapters can be used to construct mechanisms specifically aimed at tackling performance failures.

Finally, in Chapter 6, we boil down the previously presented solutions to derive protocols specialized for the single back-end database scenario.

Most of the material presented in this document can also be found in the following technical articles:

F. Quaglia and P. Romano,
"Ensuring e-Transaction with Asynchronous and Uncoordinated Application Server Replicas", *IEEE Transactions on Parallel and Distributed Systems*, to appear.

P. Romano, F. Quaglia and B. Ciciani,
"A Lightweight and Scalable e-Transaction Protocol for Three-Tier Systems with Centralized Back-End Database", *IEEE Transactions on Knowledge and Data Engineering* , vol.17, no.11, pp.1578-1583, 2005.

P. Romano and F. Quaglia,
"Providing e-Transaction Guarantees in Asynchronous Systems with Inaccurate Failure Detection", *Proc. 5th IEEE International Symposium on Network Computing and Applications (NCA)*, IEEE Computer Society Press, July 2006,

P. Romano, F. Quaglia and B. Ciciani,
"Design and Evaluation of a Parallel Edge Server Invocation Protocol for Transactional Applications over the Web", *Proc. 6th IEEE Symposium on Applications and the Internet (SAINT)*, IEEE Computer Society Press, January 2006.

P. Romano, F. Quaglia and B. Ciciani,
"A Simulation Study of the Effects of Multi-path Approaches in e-Commerce Applications", *Proc. 11th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS)*, IEEE Computer Society Press, April 2006.

F. Quaglia and P. Romano,
"Reliability in Three-Tier Systems without Application Server Coordination and Persistent Message Queues", *Proc. 20th Annual ACM-SIGAPP Symposium on Applied Computing (SAC)*, ACM Press, March 2005.

P. Romano, F. Quaglia and B. Ciciani,
"Design and Analysis of an e-Transaction Protocol Tailored for OCC", *Proc. 5th IEEE Symposium on Applications and the Internet (SAINT)*, IEEE Computer Society Press, January/February 2005.

P. Romano and F. Quaglia,
"A Path-Diversity Protocol for the Invocation of Distributed Transactions over the Web", *Proc. IEEE International Conference on Networking and Services (ICNS)*, IEEE Computer Society Press, October 2005

P. Romano, F. Quaglia and B. Ciciani,
"A Protocol for Improved User Perceived QoS in Web Transactional Applications", *Proc. 3rd IEEE International Symposium on Network Computing and Applications (NCA)*, IEEE Computer Society Press, August/September 2004.

P. Romano, F. Quaglia and B. Ciciani,
"Ensuring e-Transaction Through a Lightweight Protocol for Centralized Backend Database", *Proc. 2nd International Symposium on Parallel and Distributed Processing and Applications (ISPA)*, LNCS, Springer-Verlang, December 2004.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last decade we have witnessed the extraordinary growth of the Internet: in the early nineties, Internet users were mainly researchers from academy and industry; since the birth of the World Wide Web, the Internet has become a mass communication medium and an inexhaustible source of information. Actually the deep evolution that affected the Web over the last years is not *only* limited to the drastic increase of the traffic volume and of the number of users: a variety of factors have induced a radical change in the Web, opening a number of challenging research issues. The most important aspect of such an evolution is probably the change in the nature of the available services: indeed, the Web has moved from a simple communication and browsing infrastructure for retrieving static information to an ubiquitous medium enabling a huge variety of complex and critical electronic services, among which electronic payment systems, real-time stock/equity trading applications and on-line auctions represent just a few remarkable examples. A characterizing feature of such emerging Internet services is that, due to the critical nature of the managed data and to the urge of enforcing its consistency despite the concurrent interactions of a vast number of users, they have been growingly relying on transactional technologies, at the point that nowadays database management systems (DBMS) have de facto become an essential, omnipresent component of whatever complex Web-based application.

The increasing trend of integrating Web-based and transactional technologies has also determined the widespread diffusion of three-tier, and in general multi-tier, systems. By reflecting the logical decomposition of applications (e.g. presentation, logic, and data in a three-tier system) at both the software and hardware level, these systems represent the natural architectural support for transactional Web-based applications. In this class of applications, in fact, it is typical for middle-tier Web/application servers to endorse the responsibility to interact with back-end databases on behalf of the client (e.g. an

applet running in a browser), since the latter, for a number of reasons ranging from security to performance, is normally prevented from directly accessing the DBMSs hosting the application data.

1.1 Reliability Issues in Multi-Tier Systems

A direct consequence of the increasing diffusion of highly critical Internet services has also been the fostering of an intense research activity in the area of reliability of Web-based applications. In fact, it is nowadays widely recognized that reliability represents a critical issue for a large and growing part of Web-based transactional systems, as in the case of commercial or financial applications, in which system inconsistency due to the occurrence of failures, may imply a loss of revenue or customer loyalty and may even have legal or ethical implications.

Concerning this point, even though the partitioning of an application into multiple tiers provides the potentialities to achieve high modularity and flexibility, the multiplicity of the employed components, and their interdependencies, make reliability a not trivial issue to tackle [45, 44] and raise the need for novel, ad-hoc solutions. As an example, in database systems, which form the backbone of mission critical business applications, recovery techniques are traditionally based on the fundamental notion of atomic transactions [52, 17], providing "all-or-nothing" guarantees on the execution of correlated data manipulation statements. However, all-or-nothing transactional guarantees provided by the back-end tier are insufficient when considering the wide spectrum of failure scenarios that can arise in a multi-tier system. The major limitation of those solutions is precisely the impossibility for the client-side software to accurately distinguish the "all" from the "nothing" scenario. If a failure occurs at the middle or back-end tiers during request processing, or a timeout period expires at the client side, the end-user typically receives an exception notification. This does not convey what had actually happened, or whether a new state was actually stored in the database. In such scenarios the application program or user cannot blindly re-initiate a transaction as the transaction may have succeeded and re-execution is not usually idempotent. At this end, a common approach (although hazardous for a wide set of applications) is to warn users not to hit the checkout/buy/commit button twice when a long delay occurs. However, users who do not heed this warning may, e.g., unintentionally purchase two seats on the same flight or two copies of the same book. On the other hand, an even worse situation arises if a failure occurs without providing any notice. For an e-Commerce service, for example, failures occurring during shopping cart checkout can lead to user inconvenience and lost sales.

Concerning Internet services, one main feature having a deep impact on reliability aspects is that, if on one hand the Internet has the amazing potentiality to provide access to information and services spread on planetary scale, on the other hand its architecture is still mainly oriented toward a best-effort model. This currently precludes the possibility to provide any guarantee either on the communication delays, or on the probability that a service residing at some host becomes temporarily unreachable due, e.g., to temporary network congestions/failures. This exposes the Internet's ultimate clients, which possibly also include the server-side components of a Web-based application, to the threats raised by large, highly fluctuating and typically unpredictable communication latencies and to the anomalies related to the inherent asynchrony of the system, which, on their turn, may affect the frequency of, e.g., heisenbugs [50] manifestation due to race conditions. In addition, recent advances in hardware and software technologies have created a plethora of mobile devices, enabling a challenging scenario of ubiquitous mobile computing that keeps people connected to Internet applications and services at all times, regardless of their location or access device. Unfortunately, mobility introduces a number of additional problems. For example, mobile client devices have typically limited communication, computing and storage capabilities, are subject to power consumption constraints and are usually forced to operate in relatively less reliable environments.

1.2 Key Design Choices Impacting Reliability Issues

In the wide spectrum of possible design choice characterizing modern multi-tier systems it is possible to identify a few, high level system features which, independently of the specific technologies actually employed (e.g. J2EE or .NET as the middle-tier middleware platform), have a deep impact on the viable solutions for tackling relevant (end-to-end) reliability issues. We shortly discuss these features in the next sections in order to provide a base ground functional to the introduction of concepts that will help clarifying reasoning and will allow to better frame the contributions of this thesis.

1.2.1 Statefulness vs Statelessness of Middle-Tier Servers

In the context of traditional client/server systems, the notion of "state" is usually defined as any data that can be affected by a client request. Multi-tiered applications deal with two types of state: hard and soft.

Hard state is what cannot be reconstructed easily or at all, hence the hard state is typically entirely stored in the back-end databases to guarantee persis-

tence and durability. Examples of hard state are: stock information about the available books in an on-line book store, the email messages received by a user of an email service, or the highest bid for each item available in an auction service. Soft state is what can be easily reconstructed, either automatically or with user help, and so does not need to be persistent. Web applications deal with a variety of soft states, such as user profiles, navigation tracking information, the contents of HTML forms, etcetera. An other relevant example of soft state is the so called session state. A session contains information about a series of sequential requests from a single user (e.g., the contents of a shopping cart prior to the checkout interaction). The common approach is to assume that a session is over if the user does not access the service for some time (e.g., 30 minutes): in such a case the corresponding state is simply discarded [46].

Even though storing the soft state within the back-end databases remains a feasible solution, given that soft state typically does not require full ACID semantics, two additional strategies can be adopted for its maintenance. It can either be stored at the client (encoded as cookies, or as parameters in URLs, or as hidden fields of HTML forms) and piggybacked on the messages exchanged between the client and the application server. Alternatively, the soft state can be maintained by the middle-tier application server, typically by means of in-memory session-objects. In the latter case, application servers are said to be *stateful*. Conversely, if application servers do not maintain any soft state across different client requests, they are said to be *stateless*. In general, the two approaches are both widely employed in current multi-tier applications and show often complementary advantages and drawbacks.

The main advantage provided by the assumption of stateful middle-tier servers is in that it is, generally speaking, less restrictive. Also, it cannot be avoided in certain scenarios. This happens, for example, if the middle-tier components actually store part of the application hard state (e.g. like in the case of some complex workflow services) or in case some constraints exist which preclude the possibility of delegating the maintenance of the middle-tier (soft) state to some other tier (e.g. due to restrictive security or privacy policies). Another advantage of relying on middle-tier application servers for storing the application soft state is that it allows offloading the back-end database, which is one of the potential system bottlenecks. This may result in an improvement of system throughput especially if soft state is large and needs to be frequently updated.

On the other hand, stateless application servers, having no client affinity ([1]), make the load balancing mechanism drastically more efficient and flexible. First, the computational load on the load balancing component (which,

[1]The client requests can be forwarded to any stateless application server, rather than to the application server that holds the corresponding client state.

as shown by a number of recent studies [5, 23, 68], can easily become the bottleneck in high performance Web architectures) is strongly reduced. Further, it allows to react more effectively and promptly to possible load unbalancing situations [105], by simply derouting incoming traffic from overloaded servers to less loaded ones. Also, a recent work [46] has demonstrated that maintaining soft state at the back-end tier normally achieves better performability than storing it in the middle-tier servers' main memory. In particular, it has been shown that most of the unavailabilities are typically due to faults in the middle-tier, and that strategies maintaining soft state at the middle-tier increase the impact of faults in these tiers to such an extent that, overall, availability is degraded by a larger factor than performance is improved by offloading the back-end databases.

Always concerning reliability, in case of stateless servers, the fail-over of a crashed application server to a working replica can be more simply and promptly performed, since no state transfer mechanisms must be adopted. Additionally, stateless application servers must not wait for state recovery upon restart (e.g. after a failure), which reduces the duration of downtimes and increases perceived availability. Also, from a reliability standpoint, the crash of the single (stateless) application server contacted by the client is equivalent, to the crash of any of the (stateless) application servers in a chained invocation scheme [45, 44], hence permitting to model, without loss of generality, a general multi-tier system as a three-tier system. This considerably simplifies the reasoning on the fault-tolerance properties of the system.

A final important consideration is that recent advances in middleware technology have opened the possibility of automatically transforming middle-tier stateful components into stateless entities, thus increasing the generality of the latter approach. The widely adopted approach to delegate commonly required lower level functionalities, such as persistence of the application state, to container environments (e.g., Sun Microsystems J2EE [114]) provide developers with the ability to control the strategy employed for maintaining the soft state in a simple declarative way by means of extended deployment descriptors. In other words, application designers can transparently configure the hosting middleware platform so to avoid maintaining soft state in the middle-tier servers' memory and resort to the employment of one of the aforementioned alternative strategies (e.g. encoding it within the response to the client or storing it within a back-end database).

1.2.2 Deployment of System Components

Within the design space of a multi-tier transactional application, there are a number viable alternatives for what concerns the deployment strategy of the various application components, i.e. the allocation of the individual appli-

cation sub-systems to a finite set of physical computing, communication and storage resources. In this context, a fundamental parameter impacting reliability issues is certainly the scale, ranging from *local* to *geographical*, of the underlying system infrastructure over which the application components are deployed.

Another relevant parameter deeply affecting to what extent a given application can be controlled and configured is whether the adopted infrastructure is directly owned (e.g. a private cluster of workstations), or is rented from a third-party provider with whom certain service levels may have been agreed (e.g. a Web hosting service), or is rather a public shared utility (e.g. an Internet public communication link).

In a complex multi-tier application one can envision a quite large number of alternative configurations, but the three most representative settings that can be found in practice are probably the following ones:

- *All the system components are deployed over the same local area infrastructure:* applications adopting such a configuration are typically referred to as Intranet applications. These are widely employed within closed, private environments, e.g. internal enterprise information systems, where it is normally possible to accurately monitor and control not only the server-side infrastructure but also the client population, thus making it relatively easy to achieve desired reliability (and performance) levels.

- *Geographically distributed clients, middle and back-end tiers deployed over the same local area infrastructure:* this configuration is very common in case of Web sites relying on, e.g., a clustered architecture to host the whole set of server-side components. In such a setting, recent advances in cluster technologies have provided powerful means to pursue high reliability (and performance) levels on the server-side of the application. Compared to the previous configuration, in this scenario the main problem is how to provide supports for reliability on the whole end-to-end chain of components, given that the interactions between the clients and the server side can take place over an open and intrinsically unreliable medium such as the Internet.

- *All the system components, including middle and back-end tier components, interact over a geographical scale infrastructure:* such a configuration is found, generally speaking, in the case of middle-tier business logics relying on some remote data sources. A particularly interesting instance of such a configuration is represented by the emerging large scale service delivery platforms, also referred to as Application Delivery Networks (ADNs). ADNs provide third party application providers with an

increasingly large number of servers distributed at the edge of the network (the largest service delivery platform, currently owned by Akamai, is composed by more than 20.000 servers spread in 2.500 different worldwide locations), offering replicated access points to their applications' contents and business logics ([2]). ADNs are highly attractive infrastructures, given that they have the potentialities to achieve high scalability levels and reductions in the perceived response times thanks to their unparalleled redundancy and increased client proximity. However, applications hosted by ADNs (for a variety of reasons ranging from security or pragmatical constraints, to well known scalability/performance issues proper of transactional replication schemes over WAN [20, 51]) typically rely on remote interactions with third party back-end data sources when processing requests that involve transactional manipulations (i.e. updates) of the application hard state.

Hence, in comparison with the previously described components deployment strategy, we are here faced with the additional challenge of addressing reliability in the interactions between server side components (e.g. middle-tier serves and back-end databases), when considering that these interactions take place via an infrastructure possibly layered on public networks over the Internet, and possibly belonging to (and controlled by) providers offering different guarantees on service levels. This dramatically reduces the level of synchrony and control (e.g. fault monitoring capabilities) across the whole system. In particular, the application components (including application servers and back-end databases) get much more loosely coupled than in the previously considered configurations, which further complicates the design of reliability solutions and protocols.

1.3 Multi-Tier Systems vs Distributed System Models

In order to simplify the reasoning about properties of multi-tier systems and to abstract unnecessary details such as specific technological issues, it is convenient to encapsulate the characterizing aspects of this relevant class of distributed systems (e.g., those related to the different infrastructural alternatives) within formal distributed system models which are both accurate and tractable.

[2] ADNs represent the natural evolution of classical Content Delivery Networks [36, 121], where edge servers have exclusively the functionality to host static contents (e.g. html pages, images, videos).

Even tough a large number of different distributed systems models have been proposed over the last 20 years, certain aspects of models appear repeatedly in the literature [48]. For example, processes are usually modeled as state machines (whose behavior being possibly described by means of pseudo-code) that in turn are used to model the execution of the whole distributed system. Other aspects comprise assumptions about the network topology, the reliability of the communication channels or the available message exchange primitives.

Overall, existing models of distributed systems differ in one particularly important aspect, their inherent notion of synchrony. This is usually expressed by means of a set assumptions about process execution speeds and message delivery delays. In synchronous systems [78], the existence of bounds on message transmission delays and process response times is assumed. If no such assumptions are made, the system is called asynchronous [66, 41]. Going back to the discussion of system components deployment, the notion of asynchronous system could be used, for example, to reflect very realistically the inherent aspects of large scale multi-tier systems with loosely coupled components (like in the case of components owned and controlled by providers not offering mutual guarantees on processing and communication speed).

It is well known that the asynchronous model is the weakest one, implying that every distributed protocol that works in the asynchronous model also works in all other models. On the other hand, algorithms for synchronous systems are prone to incorrect behavior in case even a single timing constraint is violated. This is why the asynchronous model is so attractive and has attained so much interest in distributed systems theory. Also, highly variable workloads on network nodes belonging to multiple autonomous systems make reasoning based on time and timeouts a delicate and error-prone undertaking. All these facts are sources of asynchrony and contribute to the practical appeal of having few or, better, no synchrony assumptions.

However, the asynchronous model has severe drawbacks, especially for fault-tolerance applications. For example, it can be shown [26] that in such models it is impossible to (accurately) distinguish a correct process from a faulty one. Intuitively, this is a result of allowing processes to be arbitrarily slow [25].

The impossibility results afflicting asynchronous systems have fostered the development of intermediate models based on the idea of adding some synchrony assumptions to the model; these are often called partially synchronous. The original work on partial synchrony [38] assumes that upper bounds on message delivery delay or relative processor speeds exist, but they are either unknown or only hold eventually. Other prominent models are the timed asynchronous model [32] and the quasi-synchronous model [2]. The former mainly assumes a bounded drift rate of local hardware clocks while the latter

postulates that only part of the network is truly synchronous.

An alternative approach is the one undertaken by Chandra and Toueg in [25] which propose a modular way of extending the asynchronous model to detect process crashes. In their theory of unreliable failure detectors, they propose a program module that acts as an unreliable oracle on the functional states of neighboring processes. The main property of failure detectors is their *accuracy*: in its weakest form, a failure detector will never suspect at least one correct process of having crashed. This property is called weak accuracy. Because weak accuracy is often difficult to achieve, it is often required only that the property eventually holds. Thus, an eventually weak failure detector may suspect every process at one time or another, but there is a time after which some correct process is no longer suspected. In effect, an eventually weak failure detector may make infinitely many mistakes in predicting the functional states of processes, but it is guaranteed to stop making mistakes when referring to at least one process. Requiring weak accuracy alone, however, doesn't ensure that a crashed node is suspected at all: it merely prohibits the detection mechanism from wrongly suspecting a correct node. So a second property of failure detectors is necessary. Chandra and Toueg call it *completeness*. Informally, completeness requires that every process that crashes is eventually suspected by some correct process.

Obviously, models using unreliable failure detectors are no longer truly asynchronous, given that, as we have already mentioned, it is not possible to devise implementations of a failure detector providing accuracy guarantees in an asynchronous system; they merely produce the illusion of an asynchronous system by encapsulating all references to time. The relevance of this model is, on one hand, that it can be shown that different forms of unreliable failure detectors are sufficient to solve fundamental problems which were proved to be unsolvable in asynchronous systems [25, 101, 102, 98], and, on the other, that failure detectors simplify the task of designing algorithms for asynchronous systems at large, because they encapsulate the notion of time neatly within a program module.

1.4 Thesis Focus and Key Contributions

This thesis presents the results of a study addressing reliability issues in multi-tier transactional systems. We focus on the scenario of stateless middle-tier servers, which as already hinted in Section 1.2.1 is representative of a relevant and widely adopted class of modern multi-tier systems.

As we will detailedly discuss in Chapter 2, where a literature overview on reliability approaches is provided, the problem of how to support meaningful end-to-end reliability guarantees in a multi-tier system with stateless middle-

tier serves has been formalized by Frolund and Guerraoui [45, 43, 44] through the *e-Transaction* (exactly-once Transaction) abstraction. This abstraction is the main frame within which the result of this thesis can be collocated.

Actually, existing solutions to the e-Transaction problem fail to cover the whole spectrum of alternatives proper of the design space of a multi-tier transactional application. More specifically, we will show that, when considering the emerging, attractive scenario of application components deployed over large scale infrastructures, those solutions either provide poor (or suboptimal) performance levels, or result not viable, e.g. due to their reliance on synchrony assumptions which may not be supported in practice by the underlying system. We overcome these limitations by providing solutions which are effective and viable in large scale infrastructures (possibly) not exhibiting those synchrony features. In more detail:

- The first e-Transaction protocol we present (see Chapter 3) works in asynchronous systems with unreliable failure detection (i.e. an eventually weak failure detector) and, for the general case of multiple, autonomous back-end databases, is the first e-Transaction solution in literature which avoids coordination based approaches at the middle-tier servers, thus achieving unparalleled scalability level. Also, this protocol directly complies with conventional distributed transaction management technology.

- The second e-Transaction protocol we present (see Chapter 4) further enhances the latter result since it tackles the very same case, namely multiple, autonomous back-end databases, but does not rely on any accuracy assumption on the underlying failure detection mechanism, thus representing the first e-Transaction protocol suitable for a purely asynchronous distributed system model. To achieve such a target, this protocol exploits an innovative scheme for distributed transaction management, requiring ad-hoc demarcation and concurrency control mechanisms, which we introduce in this thesis. However, possible integration with conventional technology (e.g., database systems) is also discussed.

As a matter of fact, the avoidance of accuracy requirements concerning the failure detection mechanism, which is an essential component and strength point of the protocol presented in Chapter 4, can provide a new approach for jointly addressing both reliability and performance issues, especially when considering the case of large scale infrastructures. In particular, we discuss how the ability of our e-Transaction protocol to achieve real concurrency while processing multiple instances of the same client request (originated by, possibly false, failure suspicions) can be used to derive a parallel invocation protocol (to our knowledge never proposed in the context of transactional requests over

multi-tier systems) allowing to effectively cope with performance failures by exploiting both the inherent parallelism of large scale service delivery platforms and the path diversity provided by current multi-hop network infrastructures. We show that the benefits achievable by such parallel invocation schemes are twofold. First, they attack the large fluctuations in the message delivery latencies that affect Internet-based communications [62, 76, 75], with direct benefits on the average user perceived response times. Furthermore, they provide for seamless failure masking by avoiding the inherent latency of failure detection mechanisms.

The results related to this part of the thesis are all reported in Chapter 5.

As hinted above, all the previous results deal with the general context of applications characterized by business logics prescribing the manipulation of data hosted by multiple, autonomous, distributed back-end databases, as in the case of multiple parties involved within a same business process. Anyway, in real life, we can also find a wide range of application domains requiring transactional interactions with a single, centralized back-end database.

We discuss this specific scenario in Chapter 6, showing how the previous solutions could be boiled down and specialized for this simpler, yet very common scenario.

Chapter 2

Literature Overview

Over the years the reliability problem in multi-tier systems in presence of crash failures (i.e. not byzantine [40, 37, 67]) has been addressed by both the database and the distributed systems communities. As it often happens when multiple research communities having diverse scientific backgrounds and biases investigate the same problem, these two communities ended up tackling the reliability problem from different perspectives and adding emphasis on different, yet typically correlated, aspects. The distributed systems community has mainly focused on the problem of how to ensure consistent integration between the (middle-tier) replication strategies (commonly employed to pursue high availability and fault-tolerance) and the classic transactional technologies proper of back-end databases. On the other hand, authors from the database systems community have rather oriented their investigations to identify the minimal logging requirements to achieve desirable (end-to-end) reliability guarantees, without however explicitly taking into account the presence of replicated components across the invocation chain (particularly at the middle-tier) and the consequent need to ensure high availability of the logged recovery information so to provide replicas with mutual fail-over capabilities.

The choice of whether to explicitly consider the presence of replication schemes when addressing reliability issues has pros and cons. Not explicitly considering replication strategies for the components typically allows keeping both the developed reliability solutions and the reasoning on their correctness relatively simpler. However, it also exposes such solutions to the presence of single points of failure, making the liveness dependent on the availability of the recovery information logged across the various components. For these reasons, in practical settings, some additional form of replication/redundancy of the logged recovery information is highly desirable especially on the server side part of the application. In this sense, the choice of explicitly modeling the presence of replicated components along the invocation chain allows to

pursue a tighter integration between the logging strategy and the underlying replication scheme, opening the possibility for further optimizations.

As already hinted in Chapter 1 (see Section 1.2.1), another fundamental aspect that deeply affects the way in which reliability issues have to be addressed in a multi-tier system is whether middle-tier servers are stateful or stateless entities. In the former case, one needs to ensure the mutual consistency of the middle-tier servers and the back-end database servers states in the presence of failures. In the latter one the attention is instead devoted only to ensure the consistency of back-end databases, in terms of atomicity of the distributed transaction and avoidance of transaction duplication in case of failures.

This chapter surveys existing literature on reliability in multi-tier systems. We first review solutions addressing end-to-end reliability issues for the scenarios of both stateless and stateful middle-tier servers. Then we proceed with a brief overview of other reliability approaches, which can not be easily framed within any of the two aforementioned categories (mostly because they do not cope with end-to-end reliability arguments, but simply with reliability issues for a subset of the components involved in the end-to-end interaction). Finally, a discussion on literature solutions is provided.

2.1 End-to-end Reliability Approaches

2.1.1 The Case of Stateless Middle-Tier Servers

As hinted in Section 1.2.1, in the case of stateless middle-tier servers, chained server invocation over the middle-tier does not present additional reliability challenges [43, 44, 45]. Hence it is possible to abstract over chained invocation by considering a classic three-tier system rather than a general system comprising an arbitrary number of tiers. For three-tier systems, a widely adopted end-to-end reliability solution relies on queued transactions, invented for OLTP (on-line transaction processing) [16, 52, 17, 13] and supported by most TP monitors (e.g., IBM MQ Series, BEA Tuxedo, Microsoft MTS) and by the associated Web application servers (e.g., IBM WebSphere, BEA WebLogic, Microsoft IIS). Messages between clients, the TP monitor acting as an application server, and the database servers are held in transactional message queues whose operations (message enqueuing or dequeuing) are embedded in a distributed atomic transaction [15, 98] together with its all or nothing guarantee. The queue manager usually is a separate resource manager with its own log and needs to support the two-phase commit protocol (2PC), see, e.g., [17], for distributed transactions that access both messages in queues and permanent databases. An application program can then read an input message from a queue, process the message by querying and updating one or more

databases, and finally place a reply message into a queue - all in one atomic transaction. Should the database transaction abort, the message that was originally dequeued from the input queue is automatically placed back into the queue (based on log entries allowing to correctly undo the transaction), so that the input message will eventually be processed even after an arbitrary number of failures. Further, the message will be processed exactly once (for the commit of the transaction also commits the dequeuing of the input message and the enqueuing of the output message). We note that this approach involves three transactions per user request (to enqueue the request on the queue; to dequeue it, process it in the database servers, and enqueue the reply; and to dequeue the reply) which is the reason why this solution is adopted especially in the case of applications admitting off-line request processing.

The work in [70] provides end-to-end transactional integrity across the whole end-to-end interaction by encapsulating within the same atomic distributed transaction both processing and the storage of the outcome at the client. This solution exploits Java technology to empower Web browsers so that they can be viewed as a recoverable resource participating in a 2PC protocol coordinated by the middle-tier application server. At this end, the interfaces defined by the OMG's Object Transaction System [90] are exploited to implement a proxy object executing within the browser sandbox so to ensure persistency of the application response message, encoded as a cookie residing on the client disk, and its consistent manipulation according to the indications of the 2PC coordinator. By involving the client within the 2PC protocol mutual consistency of the state at the client and at the back-end databases can be easily ensured since we have guarantees that the response is delivered to the client only if the server-side transaction has successfully committed. On the other hand, the requirement for persistence capabilities at the client side, make this solution not viable in the case of browsers not having access to disk for security issues or in the case of clients having constraints on the available hardware resources (e.g. cellphones or PDAs).

For three-tier systems with stateless application servers, Frolund and Guerraoui have recently proposed the e-Transaction (exactly-once Transaction) abstraction as a desirable set of reliability properties [45]. This abstraction provides a clear-cut specification for the design of reliability protocols in this kind of systems. The e-Transaction abstraction is formally defined by three categories of properties: Termination, Agreement, and Validity. Termination properties ensure the liveness of the client-initiated interaction from a twofold prospective: not only it is guaranteed that a client does not remain indefinitely blocked waiting for a response, but also that no database server maintains precommitted data locked for an arbitrarily long time interval, no matter what happens to the client. The latter requirement is clearly very important because the availability of data (and consequently of the whole application) may

be compromised if a database server that has endorsed the responsibility to commit a transaction (i.e. the transaction is in the pre-commit state), would not timely abort or commit it. Furthermore, it prevents undesirable scenarios in which data availability gets compromised by frequently disconnecting clients, either maliciously or because of the inherent unreliability of their network connectivity (as in the case of mobile clients). Agreement embodies the safety properties of the system, ensuring both atomicity and consistency of the distributed transaction, and at-most once semantic for the processing of client requests. Finally, Validity restricts the space of possible results to exclude meaningless ones, e.g. where results are invented or transactions are committed even though some database is unable to pre-commit them.

Along with the e-Transaction specification, Frolund and Guerraoui also proposed a set of distributed protocols ensuring the e-Transaction properties. One common feature to all these protocols is that the middle-tier comprises a set of replicated application servers. Hence these solutions all rely on an explicit notion of replication at the level of middle-tier components. The e-Transaction protocol in [45], which also inspired a number of other works such as [77], is essentially based on the combination of a distributed commit scheme, namely 2PC, with a primary-backup replication approach [21, 96]. Basically, the client retransmits the request to the application servers until it receives back a result. Along the lines of classic primary-backup replication algorithms, the primary application server, before returning the response to the client, makes sure (via explicit acknowledgment) that the backups have received both (1) the result deriving from the non-deterministic interaction with the back-end databases and (2) the outcome of the vote phase of the distributed commit protocol. The primary-backup approach ensures high availability of both the transaction outcome (commit/abort) and the result to be delivered to the client. This is crucial to allow backup replicas to correctly perform fail-over in case of failure of the primary. As in classical primary-backup schemes, the protocol tolerates crashes of all except one application server. The correctness of this protocol is based on the assumption of perfect knowledge about failures among the application servers (i.e. Perfect Failure Detection in the sense of [25]), meaning that the following properties needs to be satisfied:

- *Completeness.* If any application server crashes at time t, then there is a time $t' > t$ after which it is permanently suspected by every application server.

- *Accuracy.* No correct application server is ever suspected by any application server.

The protocol in [44] relies on an asynchronous application server replication scheme. The protocol tolerates the crashes of a minority of the application

server replicas, and does not preclude the possibility of false failure suspicions (e.g. due to slow messages or network partitions). It rather assumes that failure detection among application servers is Eventually Perfect in the sense of [25], meaning that the following properties are satisfied:

- *Completeness.* If any application server crashes at time t, then there is a time $t' > t$ after which it is permanently suspected by every application server.
- *Accuracy.* There is a time after which no correct application server is ever suspected by any application server.

To deal with the possibility of various replicas performing transactions on behalf of the same client request (because of false failure suspicions) and to shelter from the inherent non-determinism of the interaction with third-party databases, this protocol employs a consensus-based [41, 98] coordination scheme among application servers. More in detail, this solution relies on a consensus abstraction referred to as Write-Once Register, which is used with the twofold purpose of (1) ensuring high availability of the transaction processing state and of the response message to be delivered to the client, and of (2) synchronizing the application servers possibly activated on behalf of the same client request.

The same authors have also presented an e-Transaction protocol [43] specifically tailored to the simpler scenario of applications interacting with a single, centralized back-end database, as in the case of a business process involving a single data source, instead of the more general case of multiple, autonomous sources. This protocol is based on the so called "testable transaction" abstraction, whose key idea is to leave a persistent trace of transaction execution in the database so to permit (1) testability of the transaction outcome and (2) retrieval of the non-deterministic result produced by the execution of the transactional business logic. This protocol handles failure suspicions through a "termination" phase executed upon timeout expiration at the client side. During this phase, the client sends, on a timeout basis, terminate messages to the application servers in the attempt to discover whether the transaction associated with the last issued request was actually committed. An application server that receives a terminate message from the client tries to rollback the corresponding transaction, in case it were still uncommitted (possibly because of the crash of the application server originally taking care of it). At this point the application server determines whether the transaction was already committed by exploiting the testable transaction abstraction. In the positive case, the application server retrieves the transaction result to be sent to the client. Otherwise, a rollback indication is returned to the client in order to allow it to safely retransmit a new request message to whichever application

server. The protocol's correctness depends on two main assumptions: (i) that a request message is always processed by the database before the corresponding terminate messages, and (ii) that the client's failure detector is Eventually Perfect.

2.1.2 The Case of Stateful Middle-Tier Servers

Similarly to the case of stateless middle-tier servers, the literature on end-to-end reliability for multi-tier systems with stateful servers can be broadly framed depending on whether middle-tier server replication is explicitly considered or not.

The works in [13, 73, 11], which extend and generalize the client-server tailored solution in [74], focuses on the design of optimized logging schemes for ensuring correct recovery of components after failures, rather than addressing the problem of how to exploit replication to achieve high availability of the application state and of the recovery information. These solutions distinguish three types of components: persistent components, or PCOMs, whose state should persist across failures, transactional components, or TCOMs, usually data servers, that provide all-or-nothing guarantees for atomic transactions, and external components, or XCOMs, which are used to capture human users who usually do not provide any recovery guarantees. These components are assumed to be piece-wise deterministic (PWD). A PWD component is deterministic between two successive messages from other components, so that it can be replayed from an earlier state if it is fed the original messages. The PWD assumption is not guaranteed without some effort, namely the logging of every non-determinism source, such as message receive order, interleaved access to shared data or interrupts that prompt component execution at arbitrary points. The component's logs are scanned during restart after a failure to ensure correct recovery. The recovery system intercepts all messages or other non-deterministic events; information is reconstructed from the corresponding log entry and fed to the component in place of the event. When log entries do not contain message contents, communication with the sender is required to obtain the contents. For this, an interaction contract with the sender ensures that the message can indeed be provided again.

An interaction contract specifies the joint behavior of two interacting components in the presence of failures of one or even both of them. An interaction contract requires each of them to provide certain guarantees, depending on the nature of the contract and components. Four distinct contracts are proposed: Committed Interaction Contracts (CICs), Immediately Committed Interaction Contracts (ICICs), External Interaction Contracts (XICs) and Transactional Interaction Contracts (TICs). A CIC involves a pair of PCOMs. The sender of a message ensures the persistence (via forced logging) of its state

and of the sent message, which has to be marked with a unique identifier to allow duplicate elimination. Furthermore the sender provides guarantees on its ability to resend the original message, until receiver releases it from its obligation. On the other hand, the receiver promises to eliminate duplicate messages. ICICs strengthen the CICs by having the receiver immediately force logging the received message. This clearly adds overhead and may have an adverse impact on throughput, but is desirable to promptly release a sender from its obligations and to enable the receiver to recover independently of the sender. External components (XCOMs), e.g. human users, are in general assumed not be able to persist their state, and hence cannot have committed interactions. Hence, the need to define External Interaction Contracts (XICs), involving a PCOM and XCOM: the PCOM subscribes to the rules for an ICIC, whereas the XCOM does not, hence not providing any guarantee neither on the replayability of its outgoing messages, nor on its ability to filter duplicate incoming messages (for example, when prompted to provide a previous input, a user does not necessarily deliver identical input). Finally, TICs involve a PCOM and a TCOM. A TCOM, analogously to the testable transaction in [43], ensures persistent testability of the transaction outcome (commit/abort). The main difference with the testable transaction implementation in [43] is that the result of the non-idempotent data manipulation is not persisted by the TCOM (namely the database server), but is rather logged by the PCOM, along with its internal state, prior to deliver the commit request to the TCOM. In [13] the authors prove that combining the above described bilateral interaction contracts into a system-wide agreement, provides the desired end-to-end guarantee to preserve consistency despite all failures with the exception of failures involving the interaction with XCOMs, due to their non-persistent and non-deterministic nature.

The results in [11, 73] further optimize the solution in [13] showing how, by imposing a number of (severe) restrictions on the admissible behaviors for a PCOM, it is possible to alleviate or even eliminate its logging requirements. This observation leads to the definition of "logless" components or LLCOMs that are stateful, like PCOMs, but that do not require a log. Essentially, admissible LLCOMs behaviors are restricted so to (1) ensure the determinism of their state transitions and of their outgoing messages, (2) avoid non-deterministic message receives (e.g. asynchronous message exchanges) and (3) forbid interactions (reads and/or writes) with external non-deterministic components outside of TCOMs, e.g. PCOMs. An LLCOM relies upon other components for logging its interactions. Whenever an LLCOM crashes or is deallocated to free up resources to enable scalability or other system management goals, it can be re-created only via complete replay of its entire execution history, starting from its initiation message. Obviously it is desirable to perform this replay quickly to increase availability and minimize system overhead.

This argues for further restricting the feasibility of LLCOMs only to components having a short lifetime. This is particularly important if one considers that external components might rely on a LLCOM for recovery after a crash.

The works in [128, 127, 43, 44, 45, 95] fundamentally differ from the above mentioned results precisely in that, beyond ensuring mutual consistency among interacting stateful components, they explicitly address the issue of how to ensure high availability of the middle tiers servers state by means of some replication scheme. In [95], the authors formalize the problem of a replicated server invoking another server. They call this a *replicated invocation*, and analyze the possible related anomalies which clearly depend on the actual replication scheme employed by the invoking server.

As it is well known [88, 104, 103, 21], the choice of the underlying replication scheme is strongly dependent on whether the replicas are assumed to adhere the PWD paradigm or not. With deterministic servers, active replication [104, 103] can be used. In such a case, any client sends, via total order multicast [35], the request to all server replicas, which process the requests in parallel. If this processing requires the invocation of another server, each replica issues exactly the same invocation, giving rise to the *duplicate requests* problem [81]. Because these invocations are identical, duplicate invocations can be detected and filtered, in order not to process them multiple times. This is done by having the replicas assign identifiers to their invocations. The result of the processing on the invoked component is valid for every replica and is multicast to them. At this point, each replica sends the reply back to the client which, generally, accepts the first one and discards the others.

Duplicate invocation filtering in the context of stateful, deterministic replicated servers is considered in [81, 86]. The work in [81] addresses transparency of the replication technique in the context of replicated invocation. The authors advocate the use of proxies to achieve transparency, for both the invocation and the reply to the invocation. Hence, a proxy is located with each client and server replica. To achieve transparency, these proxies also filter duplicate invocations and results, assuming that the clients and the actively replicated servers are deterministic.

The work in [95] also discusses the anomalies related to replicated invocation under the assumption that the replicated middle-tier server is nondeterministic. In this context, active replication is not viable, requiring alternative replication strategies, such as primary-backup schemes [21]. In such a case, [95] shows that if the primary replica invokes a server S but fails before updating the backups, then when a new primary is elected, due to replica nondeterminism, it might issue a different invocation to server S, or even choose a different server S', or not issue the invocation at all. This leaves *orphan requests* on S having a pending effect on its state. The authors address the *orphan requests* problem by having the primary middle-tier server broadcast

to its backups (1) sufficient information allowing fail-over replicas to undo the original invocation on S, and (2) updates of the replica state, as typical of classical primary-backup schemes. The authors model the invoked server as a transactional one and consider both the conventional ACID transactional model and a more relaxed one in which compensating transactions [47, 49] might be used to enforce data consistency via semantic undo. In the latter case, they show that the invoked server can immediately commit the transaction and relinquish any locked resource. On the other hand, in case the invoked server complies with the conventional ACID transactional model, this solution requires the transactional resource to provide durable guarantees on its ability to commit transactions, which implies maintaining persistent locks on accessed data items just like during the pre-commit phase of 2PC.

The protocol in [128] is conceptually very close to the latter solutions. In effect it specializes the primary-backup approach in [95] in order to implement it in a J2EE clustered environment, where stateful replicated application servers interact with a single (ACID) database server. In this solution transaction outcome testability is achieved by storing a persistent trace of the transaction execution within the database, analogously to what is done in [43]. This allows avoiding the database to pre-commit transactions, as instead required in [95].

In [127], the same authors extend the previous approach to tackle the scenarios in which multiple requests from a client run within a single transaction (N-requests/1-transaction) and/or a client request generates more than one transactions (1-request/N-transactions). To address the N-requests/1-transaction scenario the authors present two approaches, the N-1-best-effort and the N-1-ordered protocols. With N-1-best-effort the client persists all requests and corresponding responses for each transaction. If the primary crashes while a transaction was active, the client algorithm replays the execution at the new primary in a transparent way for the final user. If it leads to the same results as the original execution, fail-over is completely transparent. Conversely, if it leads to different results due to non-deterministic processing of one of the previously submitted requests, the replay is considered unsuccessful and the re-executed transaction is aborted. The end user, having seen the old non-repeatable responses, is informed with a failure exception, and hence transparency is lost. The N-1-ordered protocol attempts to improve transparency by tackling the database induced non-determinisms at the price of higher overhead during normal processing. With this approach, the re-execution of all database accesses is performed in the same order as during the original execution. During normal processing, each database access is assigned a unique increasing identifier. Before the response for the request is returned, a message with the identifiers of all access triggered by the request is multicast to the backups. At the time of resubmission after the primary crashes, each replayed database access must be executed according

to its original order and new requests may not start until all resubmissions have completed. The solution proposed to tackle the 1-request/N-transactions pattern simply extends the idea of relying on the database for filtering duplicate transactions, by keeping track within application level database tables of each committed transaction triggered by a client request. The main problem here is that, upon fail-over, non-deterministic business logic may determine the execution of a different transaction against the back-end database which might compromise data integrity. At this purpose the authors advocate the use of compensating transactions.

2.2 Other Reliability Approaches

As hinted, there are a number of reliability solutions which could be employed in multi-tier systems, which however do not address end-to-end reliability issues (i.e. the specific target of this thesis) with a holistic approach, hence coping with a subset of failure scenarios that arise in multi-tier systems. We briefly overview some of them in this section for completeness.

The works in [12, 42] leverage database server logging to mask DBMS failures to client applications (e.g. by virtualizing ODBC sessions and materializing their state as persistent database tables). These solutions, despite being originally designed for client-server applications, can be also exploited in three-tier systems, e.g. to optimize server side failure handling by masking back-end database crash and recovery to the middle-tier application server executing the transactional logic.

Another approach to address reliability in three-tier systems is the use of group communication [39, 79, 87]. However the target of this approach is to provide reliable delivery of client requests at the middle-tier, not to provide end-to-end reliability in case the middle-tier interacts with back-end databases.

Finally, concerning building blocks for PWD and replication, the work in [86] proposes an approach for enforcing determinism of multithreaded applications. This is done in the context of Eternal [86], a replication infrastructure for CORBA objects. Determinism is enforced by allowing only a single logical thread of control within each replica. Although multiple threads may exist within the replicas, all of them relate to the same logical thread of control. Consistent dispatching of threads to the replicas is achieved using a deterministic operation scheduler. Another related approach is the one in [60], where determinism of transactional multithreaded replicas is enforced in the context of active replication, even though no replicated invocation is considered. More specifically, they identify two levels of non-determinism: external and internal. External non-determinism corresponds to non-determinism related to communication, while internal non-determinism relates to computation, in particular

thread scheduling. External non-determinism is handled using totally ordered multicast. Internal non-determinism is addressed with deterministic thread scheduling and selective message reception from two-level queues.

2.3 Discussion

As mentioned in Chapter 1, the focus of this thesis is on end-to-end reliability for applications characterized by stateless application serves. Concerning previously discussed solutions dealing with such a context, there are a set of relevant issues and limitations to be addressed. In particular, the employment of these solutions in the context of large scale, geographical infrastructures (e.g. ADN like infrastructures), namely challenging and very attractive environments, raise problems of both pragmatical and theoretical nature which reduce their effectiveness, or even make them unfeasible. We discuss these problems below:

- From a pragmatic perspective, a fundamental critic that can be made to the existing solutions is the excessive cost they would impose in case of adoption on a large scale infrastructure.

  Concerning the e-Transaction solutions in [44, 45], the overhead is due to the need for executing an explicit coordination scheme among serves over the middle-tier. These coordination schemes in fact impose additional (broadcast) communication rounds among the server replicas even in absence of failure, thus hampering scalability and introducing large overhead in case of large scale, geographical distribution of the replicas. Such an overhead has also a direct impact on the end-user perceived latency since transaction processing at the application server cannot proceed until the coordination scheme does not get completed. In practice, these solutions reveal mainly attractive for scenarios where the number of application servers replicas is kept relatively low and where these replicas are hosted by, e.g., a cluster environment, where the latency of coordination can be kept low thanks to the reduced delivery latency. This is certainly not the case of large scale service delivery platforms, such as, e.g., Akamai's ADN [36], comprising tens of thousands of servers spread on global geographical scale.

  Analogous considerations also apply to the solutions in [16, 70]. For what concerns the queued transaction approach in [16], in a large scale infrastructure the queuing system is typically centralized rather than replicated at all the application servers, so to avoid of the excessive overhead for maintaining their consistence. This means that any interaction

between the application server and the queuing system implies an interaction between remote systems. Hence, the need to perform additional interactions with the queuing system (for queuing requests and responses before and after the access to the application data) translates into a considerable penalization of the end-user perceived latency. On the other hand, the approach in [70], involving the client within the transaction boundaries, requires additional communication rounds (to support the 2PC protocol) between the application server and the client. Given that, except for the case of application components deployed over a same local area infrastructure, the clients interact over relatively slower and unreliable communication channels (e.g. a GPRS or a dial-up connection), this has the twofold effect of delaying the response delivery at the client and the completion of the 2PC. This latter aspect is important since an increase in the duration of the pre-commit phase (during which back-end database servers maintain locks on valuable resources, such as application data) can lead to an increase in the likelihood of conflicts among transactions, which typically leads to strong reductions of the databases throughput [17]. We additionally note that, in a large scale, open environment, the likelihood of malicious clients intentionally delaying their replies while executing the 2PC might also be a relevant issue.

As a final additional consideration, the reliance of both the solutions in [70, 16] on 2PC introduces unnecessary overhead in the common scenario of applications interacting with a single back-end database, in which case the inherent costs of 2PC could be indeed avoided.

- From a theoretical oriented perspective, the correctness of existing solutions depends, at various extents, on assumptions regarding the synchrony of the system, or, equivalently, the accuracy of the underlying failure detection mechanism. This raises concerns on the feasibility of such approaches in the context of large scale infrastructures providing minimal guarantees, or even no guarantee at all, on the system synchrony level. In particular, the proposals in [45, 44] are based on the assumption of perfect and eventually perfect failure detection among application server replicas. Assuming perfect failure detection is clearly unrealistic in the scenario of replicas deployed over an inherently asynchronous infrastructure, such as a large scale ADN platform, where servers are distributed across multiple autonomous systems and communicate via public Internet links. This restricts in practice the feasibility of this approach to the case of application servers deployed over an infrastructure exhibiting processing/communication timing features proper of synchronous systems [38, 31] (e.g. a LAN-based infrastructure).

  The solution in [44] partially alleviates the strong synchrony require-

ments of [45], hence enlarging the spectrum of system settings covered by the protocol. Nevertheless, its reliance on an eventually perfect failure detector, to ensure the liveness of the adopted coordination scheme, does not allows the protocol to be straightforwardly adopted over whichever infrastructure where, e.g., some specific timing constraints (although relatively weak) cannot be guaranteed. Hence this protocol still does not cover the cases where the infrastructure exhibits features proper of a purely asynchronous system.

Similar considerations can be made for the e-Transaction protocol in [43] where the authors address the more limited case of a single database in the back-end tier. As we discussed, in fact, this solution relies on the assumption that a request message sent is always processed by the database before the corresponding terminate messages. Providing such a guarantee is not trivial to ensure in an asynchronous system, given that the two messages may be originated by different application replicas, upon the receipt of the corresponding client messages. To meet this assumption, the authors suggest to delay the processing of the terminate messages at the application servers, which however requires some form of timing constraints within the whole system (on both processing and communication) to achieve its goal. An alternative approach would be to rely on some form of explicit coordination among the servers (e.g. to support totally ordered multicast communication), which was not however part of the original protocol design. Nevertheless, this would impose pragmatical and theoretical drawbacks similar to the ones highlighted for the protocols in [44, 45].

Chapter 3

Ensuring e-Transaction with Uncoordinated Application Server Replicas

In this chapter we introduce an e-Transaction protocol addressing the general case of transactions spanning multiple, autonomous back-end databases (as in the case of multiple parties involved within a same business process) which does not prescribe any form of coordination among application server replicas (during both normal behavior and fail-over). The key idea of this proposal is to store recovery information concerning the transaction processing state (ITP for short - Information on Transaction Processing) across the back-end databases participating in the transaction. The ITP includes both classical information logged in standard two-phase commit (2PC) (such as the transaction state), plus additional information proper of our protocol (such as the identity of the application server processing the transaction), and allows our protocol to effectively deal with both transaction atomicity and idempotence, while preserving liveness. As a matter of fact, our approach is orthogonal to all the optimizations proposed in literature, aimed at reducing the messaging and logging overhead of standard 2PC, e.g. Presumed Commit/Abort [85], Early Prepare [111], Coordinator Log [110]. In particular, we exploit distributed logging activities performed by 2PC participants not only to achieve transaction atomicity, but also to ensure exactly-once execution semantic (i.e. idempotence and termination) of end-to-end interactions in a three-tier system. The ITP is manipulated through *local transactions*, which are independent of the global distributed transaction associated with the client request. These local transactions are autonomously executed by the database server in a transparent way for the application server, thus not requiring any additional interaction between application and database servers. This is just what allows our pro-

tocol to exhibit minimal overhead compared to a baseline approach that does not provide end-to-end reliability guarantees.

Beyond presenting the protocol, we also provide the proof of its correctness with respect to the e-Transaction properties, and the results of a quantitative comparative analysis vs other solutions carried out through parameterized performance models and an industry standard benchmark for On-Line-Transaction-Processing (OLTP) systems

3.1 Model of the System

We consider a classical distributed, asynchronous system model, in which there is no bound on message delay, clock drift or process relative speed [41]. Process communication takes place exclusively through message exchange. Processes can fail according to the crash-failure model [53]. Communication channels are assumed to be reliable, therefore each message is eventually delivered unless either the sender or the receiver crashes during the transmission. In the following paragraphs we describe the main features of every class of processes in the system, i.e. clients, application servers and database servers.

3.1.1 Clients

Client processes do not directly communicate with database servers, they only interact with application servers. This takes place by invoking the method `issue`, which is used to activate the transactional logic on the application server. This method takes the client request content as the parameter and returns only upon receipt of a positive outcome (*commit*) for the corresponding transaction. The method returns the result of the transaction execution.

3.1.2 Application Servers

We consider a set of n application server processes $\{AS_1, \ldots, AS_n\}$. Application servers collect request messages from the clients and drive updates over a set of distributed database servers within the boundaries of a global transaction [17, 98]. For presentation simplicity we assume that every transaction is executed over the same set of database servers, and that this set includes all the system back-end databases. However, in Section 3.4 we discuss approaches for relaxing such an assumption in order to cope with applications performing transactional access to a subset of the back-end databases.

Application servers have no affinity for clients, and do not know of each other's existence. Moreover, they are stateless, in the sense that they do not maintain states across requests from clients, i.e. a request from a client can only determine changes in the state of the databases.

Application servers have a primitive `compute`, which embeds the non-idempotent transactional logic for the interaction with the databases. This primitive is used to model the application business logic while abstracting the implementation details, such as SQL statements, needed to perform the data manipulations requested by the client. `compute` executes the updates on the databases inside a distributed transaction that is left uncommitted, therefore the changes applied to data are not made permanent as long as the databases do not decide positively on the outcome of the transaction. The result value returned by the primitive `compute` represents the output of the execution of the transactional logic at the databases, which must be communicated to the client. `compute` is non-deterministic as its result depends on the state of the databases, and (possibly) on other non-deterministic factors (such as the current state of some device). In other words, multiple invocations of this primitive may return different results. As in [44, 45], `compute` is assumed to be non-blocking, which means it eventually returns unless the application server crashes.

3.1.3 Database Servers

We assume a finite set of m autonomous database servers $\{DB_1, \ldots, DB_m\}$, each one possibly keeping a different data set. A database server is viewed as a stateful, autonomous resource that offers the XA interface [117] and recovers after a crash. We note that, assuming XA as the programming interface for distributed transaction management means in practice that we consider database severs directly complying with conventional transactional technology. Actually, we do not consider the whole set of functions of the XA API. Specifically, we are interested only in transaction commitment functionalities which we model through the `xa_prepare` and `xa_decide` primitives. `xa_prepare` takes a transaction identifier as input and returns a value in the domain $Vote = \{yes, no\}$. A *yes* vote implies that the database server is able to commit the transaction (i.e. the transaction is pre-committed at that database), whereas a *no* vote is returned when the database server is unable to commit the transaction (i.e. it is aborted at that database). The `xa_decide` primitive takes as input a transaction identifier and a decision in the domain $Decision = \{commit, abort\}$ and returns a value in the domain $Outcome = \{commit, abort, unknown\_tid\}$. The *unknown_tid* value is an error code reported when an unknown transaction identifier is passed as input, i.e. the database server attempts to decide on an unknown transaction ([1]). `xa_decide` returns *commit* if the database server voted *yes* for a transaction

[1] We recall that according to the XA specification a database server, i.e. a resource manager in the XA terminology, is allowed to forget about a transaction, namely about a transaction identifier, once the transaction is either committed or aborted.

and *commit* is passed as input. Otherwise the transaction is aborted and `xa_decide` returns the value *abort*.

Each database server stores some recovery information, namely the ITP (Information on Transaction Processing), which is used to determine the processing state of a given transaction. The ITP consists of (i) the identifier of the transaction, (ii) the identifier of the application server that executes the transaction, (iii) the transaction result (i.e. the output of the `compute` primitive executed by the application server), and (iv) a value that allows the identification of one of the following states for the transaction:

1. **Prepared.** This value indicates that the transaction is pre-committed at that database.

2. **Commit.** This value indicates that the transaction has already been committed at that database.

3. **Abort.** This value indicates that the transaction was aborted, or needs to be aborted, at that database.

The ITP is accessed and manipulated within ACID transactions by means of `insert`, `overwrite` and `lookup` primitives, implemented, e.g., through SQL statements on a conventional database system ([2]). `insert` takes four input parameters, namely the identifiers of the transaction and of the application server executing it, a value in the domain $\{prepared, abort\}$ and a result, and records them (i.e. inserts the corresponding tuple) within a database table. This primitive is used to mark the state of the transaction within the ITP as *prepared* or *abort*. We assume the transaction identifier to be a primary key for that database table. Therefore, any attempt to insert the previous tuple within the database multiple times is rejected by the database itself, which is able to notify the rejection event by raising the `ITPDuplicatePrimaryKeyException`. `overwrite` takes two parameters, namely the transaction identifier and a value in the domain $\{commit, abort\}$, and is used to set the state maintained by the ITP associated with that transaction identifier. More precisely, the transaction state is set to the value passed as second input parameter to the primitive. Both `insert` and `overwrite` encompass real transactional data manipulation, thus requiring eager disk access (the corresponding transactions are executed

[2] An ACID transaction manipulating the ITP can be also supported efficiently through an ad-hoc lightweight implementation exploiting capabilities of file system API, thus possibly providing performance advantages over a transaction executed on top of the database system. Actually, we have decided to describe the protocol by relying on SQL for the manipulation of the ITP exclusively for simplicity of presentation. However, as we shall discuss in Section 3.5, we have also developed an ad-hoc prototype implementation just based on file system API.

as independent top-level actions, i.e. not within the context of the transaction on whose behalf they are executing). The `lookup` primitive is used to retrieve from the ITP the state and the result associated with a certain transaction, together with the identity of the application server that executed the transaction. This primitive takes as input a single parameter, namely the transaction identifier. It returns the special value *nil* in case that transaction identifier has no corresponding ITP logged.

All the primitives available at the database server are assumed to be non-blocking, i.e. they eventually return unless the database server crashes after the invocation.

3.2 The protocol

3.2.1 Client Behavior

Figure 3.1 shows the pseudo-code defining the client behavior. Within the method `issue`, the client generates an identifier associated with the request, selects an application server and sends the request to this server, together with the identifier. It then waits for the reply, namely for an Outcome message for the transaction ([3]). If the outcome is *commit*, `issue` simply returns the result of the transaction. If the outcome is *abort*, the client chooses a new identifier and retransmits the request. Otherwise, in case the contacted application server is suspected to have crashed ([4]), the client invokes the `terminate` method. Within this method, the client keeps on retransmitting Terminate messages to the application servers (application server crash suspicion is the trigger for the retransmission), until an outcome is returned via an Outcome message indicating that the transaction was either aborted or committed. If the outcome is *abort*, the client chooses a new identifier and retransmits the request.

3.2.2 Application Server Behavior

The application server behavior is shown in Figure 3.2. It makes use of the `DTManager` class in Figure 3.3. Two execution paths are possible at the application server depending on the type of the message received from the client.

If a Request message is received, then `compute` is invoked to execute the distributed transaction. Next the application server starts the distributed

[3] We indicate with **receive** a blocking statement returning upon message delivery at the application level. Also, the **received** statement immediately returns with boolean indication on whether a given message is already available at the application level.

[4] We use the classical **suspect** statement to abstract over details of the underlying failure detection mechanism.

```
Class Client {
  CircularList aslist={AS_1,AS_2,...,AS_n};
  ApplicationServer AS;

  Result issue(RequestContent req) {
    Outcome outcome=abort; Identifier id; Result res;
    AS=aslist.next();
    while (outcome==abort) {
          set a new value for id;
          send [Request,req,id] to AS;
          wait receive [Outcome,res,outcome,id] or suspect(AS);
          if suspect(AS) (res,outcome)=terminate(id);
    }
    return res;
  } /* end issue */


  (Result,Outcome) terminate(Identifier id) {
    while (true) {
          AS=aslist.next();
          send [Terminate,id] to AS;
          wait receive [Outcome,res,outcome,id] or suspect(AS);
          if (received [Outcome,res,outcome,id]) return(res,outcome);
    }
  } /* end terminate */
}
```

Figure 3.1: Client Behavior.

commit protocol by invoking the `prepare` method of the `DTManager` class. While executing this method, the application server periodically retransmits the Prepare message on a timeout basis to all the database servers until a Vote message is received from every database server. Then, the commit protocol goes on through the `decide` method of that same class. If an unanimous positive vote was collected (i.e. every database server voted *yes*), Decide messages with the *commit* decision are sent to all the database servers. If any *no* vote was received, the whole transaction has to be aborted. In this case, Decide messages with the *abort* decision are sent to the database servers. Decide messages are retransmitted, again on the basis of a timeout mechanism, until an Outcome message is received from every database server. Once the interaction with database servers is concluded, an Outcome message is sent back to the client, carrying the outcome of the transaction (*commit* or *abort*), together with the result.

A different behavior is triggered by the arrival of a Terminate message. In this case, the application server invokes the `resolve` method of the `DTManager` class to determine the final outcome of a given transaction possibly activated by a different application server due to a Request message received from the

```
Class Application Server {
  List dblist={DB_1, DB_2, ..., DB_m};
  DTManager dt;

  void main() {
  Result res; Outcome outcome;
      while (true) {
          cobegin
            || wait receive [Request,req,id] from client;
              res=compute(req,id);
              if ( dt.prepare(id,res)==abort ) {res=nil; outcome=abort; dt.decide(id,abort);}
              else {outcome=commit; dt.decide(id,commit);}
              send [Outcome,res,outcome,id] to client;
            || wait receive [Terminate,id] from client;
              (res,outcome)=dt.resolve(id);
              send [Outcome,res,outcome,id] to client;
      }
  } /* end main */
}
```

Figure 3.2: Application Server Behavior.

client. Within the `resolve` method, the application server collects the state of the transaction logged by the ITP maintained by every database server. Specifically, it sends Resolve messages to the database servers and waits for Status messages from all of them. Also in this case we use a timeout based retransmission logic if the database servers do not respond within a pre-specified time period. If at least one database server reports an *abort* status for that transaction, the application server makes sure that the transaction is aborted everywhere. This is done by invoking the `decide` method with adequate values for the parameters. On the other hand, if every database server responds with a Status message carrying either a *prepared* or a *commit* value retrieved by the ITP associated with the transaction, the application server exploits again the `decide` method of the `DTManager` class to commit the transaction at those sites where it is prepared, but still uncommitted. Finally, the transaction outcome and the result reported by the databases are sent to the client.

3.2.3 Database Server Behavior

Figure 3.4 shows the pseudo-code defining the database server behavior. We avoid describing the database server behavior during the transaction execution phase since, as stated in Section 3.1, this phase simply encompasses, e.g., a set of conventional SQL statements which are abstracted through the application server's `compute` primitive. This server exploits the `DTManager` class in Figure 3.3 and also the `ITPLogger` class in Figure 3.5. The database server executes

```
Class DTManager {
  List dblist={DB_1, DB_2, ..., DB_m};

  (Result,Outcome) resolve (Identifier id) {
    repeat {
      send [Resolve,id] to dblist;
      set TIMEOUT;
      wait until ( ( for every DB_k ∈ dblist: receive [Status,id,status,result] ) or TIMEOUT );
    } until ( received [Status,id,status,result] from every DB_k ∈ dblist );
    if (received [Status,id,abort,nil] from some DB_k ∈ dblist)
       {this.decide(id,abort); return (nil,abort); }
    else {this.decide(id,commit); return (result,commit);}
  } /* end resolve */


  Status prepare(Identifier id, Result result) {
    repeat {
      send [Prepare, id, result] to dblist;
      set TIMEOUT;
      wait until ( (for every DB_k ∈ dblist: receive [Vote,id,vote]) or TIMEOUT );
    } until ( received [Vote,id,vote] from every DB_k ∈ dblist );
    if ( received [Vote,id,yes] from every DB_k ∈ dblist ) return prepared;
    else return abort;
  } /* end prepare */


  void decide(Identifier id, Outcome decision){
    repeat {
      send [Decide, id, decision] to dblist;
      set TIMEOUT;
      wait until ( (for every DB_k ∈ dblist: receive [Outcome,id,outcome]) or TIMEOUT );
    } until (received [Outcome,id,outcome]) from every DB_k ∈ dblist );
  } /* end decide */
}
```

Figure 3.3: `DTManager` (Distributed Transaction Manager) Class.

three tasks triggered by the receipt of different types of messages, and an additional background task.

Task 1: This task is activated upon receipt of a Prepare message from an application server. The database server invokes the `xa_prepare` primitive in the attempt to pre-commit the transaction. If this operation fails, a negative Vote is sent back to the application server. Conversely, if `xa_prepare` returns a *yes* vote, the `prepare` method of the `ITPLogger` class is invoked to insert the ITP associated with the transaction with the *prepared* state value. If the insertion of the ITP fails, it means that the database already stores the ITP associated with the transac-

```
Class Database Server {
   List dblist={DB_1, DB_2, ..., DB_n};
   DTManager dt;
   ITPLogger itp;
   Result result;
   Status status;
   Outcome outcome;

   void main(){
    on recovery do {
       for every pre-committed transaction j:
            if ((itp.retrieve(j)==abort) or (itp.retrieve(j)==nil)) xa_decide(id,abort);
    }

    while (true) {
       cobegin
          || wait receive [Prepare,id,result] from AS_i; // Task 1
            if (xa_prepare(id)==yes) vote=itp.prepare(id,result,AS_i);
            else vote=no;
            send [Vote,id,vote] to AS_i;
          || wait receive [Decide,id,decision] from AS_i or DB_i; // Task 2
            outcome=xa_decide(id,decision);
            if (outcome==unknown_tid)
                    {send [Outcome,id,decision] to AS_i or DB_i; itp.set(id,decision); }
            else
                    {send [Outcome,id,outcome] to AS_i or DB_i; itp.set(id,outcome);}
          || wait receive [Resolve,id] from AS_i or DB_i; // Task 3
            (status,result)=itp.try_abort(id);
            send [Status,id,status,result] to AS_i or DB_i;
          || background: // Task 4
            for every pre-committed transaction j such that itp.retrieve(j)==prepared:
                    if ( suspect( itp.getAS(j) ) ) dt.resolve(j);
    } /* end while */
   } /* end main */
}
```

Figure 3.4: Database Server Behavior.

tion ([5]). In this case, the transaction state maintained by the ITP is retrieved through the `lookup` primitive. If the state value is *abort*, a negative vote is sent back to the application server via a Vote message. Otherwise (i.e. the transaction is prepared and the ITP insertion with *prepared* state value succeeds) a Vote message with *yes* is sent back to the application server.

Task 2: This task is activated upon receipt of a Decide message. The database server invokes the `xa_decide` primitive to take a final decision for the

[5] We recall that we have assumed the transaction identifier to be a primary key - see Section 3.1 - therefore at most one insertion of the ITP associated with a given transaction can occur.

```
Class ITPLogger {
  Status status;
  Result res;
  ApplicationServer AS;

  Vote prepare(Identifier id, Result result, ApplicationServer AS) {
    try {
      insert(id,prepared,result,AS);
      return yes;
    }
    catch (ITPDuplicatePrimaryKeyException ex) {
      if (lookup(id).status == abort) return no;
      else return yes;
    }
  } /* end prepare */

  (Status,Result) try_abort(Identifier id){
    try {
      insert(id,abort,nil,nil);
      xa_decide(id,abort);
      return (abort,nil);
    }
    catch (ITPDuplicatePrimaryKeyException ex) {
      (status,res,AS)=lookup(id);
      return (status,res);
    }
  } /* end try_abort */

  void set(Identifier id, Outcome outcome) { overwrite(id,outcome); }

  Status retrieve(Identifier id) { return lookup(id).status; }

  ApplicationServer getAS(Identifier id) { return lookup(id).AS;}
}
```

Figure 3.5: `ITPLogger` Class.

transaction. As it will be shown by Lemma 3.3.4 in Section 3.3.2, if `xa_decide` returns *unknown_tid*, i.e. the database server is asked to decide for an unknown transaction ([6]), this implies that the database server must have already taken the same decision it is currently asked to take. Therefore, the database server simply sends back an Outcome message with an outcome equal to the requested decision, and accordingly updates the corresponding ITP. On the other hand, if `xa_decide` returns either *commit* or *abort*, the database server sends back an Outcome message carrying the outcome returned by `xa_decide` and accordingly updates the corresponding ITP.

[6] As pointed out in Section 3.1, by the XA specifications the database does not keep track of identifiers of already committed/aborted transactions.

Task 3: This task is activated upon receipt of a Resolve message. The resolve phase is used in the attempt to abort a transaction possibly left pending due to some failure (e.g. crash of the application server originally taking care of it). In this case, the database server invokes the `try_abort` method of the `ITPLogger` class in the attempt to insert the ITP with the *abort* value for the transaction state. If the latter operation succeeds, `xa_decide` is invoked during the execution of `try_abort`, with the *abort* value as input parameter, and a Status message is sent back indicating the *abort* value for the transaction state. Conversely, if the ITP insertion fails, the transaction state maintained by the ITP is retrieved through the `retrieve` method of `ITPLogger` and is sent back to the application server via a Status message.

Task 4: This is a background task used to avoid maintaining any pre-committed transaction blocked by a crash of the application server taking care of it. Actually, fail-over of a crashed application server might be performed by other application server replicas, if the client contacts them through Terminate messages. However, also the client might crash, e.g. after the issue of its request, thus originating a situation in which application server replicas are not notified that fail-over of the application server originally taking care of the transaction needs to be performed. This background task executed by the database server copes with this exact type of situation. Within this task, the database server checks whether there are transactions having an ITP with *prepared* state value, and for which the application server originally taking care of them is suspected to have crashed (recall the identity of such application server is retrieved through the ITP). For all of those transactions, the `resolve` method of the `DTManager` class is invoked to determine their final outcome.

Finally, the database server executes the following actions on recovery after a crash. Every pre-committed transaction either having an ITP with the *abort* value or having no ITP logged, is aborted through the `xa_decide` primitive. Actually, the presence of pre-committed transactions with the *abort* value within the corresponding ITP may occur due to non-atomicity in the execution of **Task 3**. Specifically, it could happen that, while executing the `try_abort` method within this task, after the successful insertion of the ITP with an *abort* indication for the transaction state, the database server is unable to execute the `xa_decide` primitive, e.g. due to a crash failure. If this problem were not tackled, we might have pre-committed transactions, which possibly hold locks on data, with an ITP indicating an *abort* state (i.e. the transactions

need to be aborted), whose abort procedure will not eventually be handled by the database server. Similarly, the presence of pre-committed transactions with no corresponding ITP logged may occur due to non-atomicity in the execution of **Task 1**, i.e. the database server might crash after preparing the transaction through `xa_prepare` but before recording the ITP through the `prepare` method of the `ITPLogger` class. These transactions can be aborted by the database server since no *yes* vote has been sent out for them (recall the *yes* vote is sent out by the database server only after successful insertion of the ITP with *prepared* value).

3.2.4 Observations

Observation 3.2.1 *By the protocol structure, a* Decide *message with the commit indication is ever sent to a database server, only if (1) positive* Vote *messages are collected by an application server from all the database servers or (2)* Status *messages with prepared/commit are collected by an application/database server from all the database servers. Note that this implies that all the database servers have performed successful insertions of the ITP with the prepared value for the transaction state.*

Observation 3.2.2 *By the protocol structure, a* Decide *message with the abort indication is ever sent to a database server, only if (1) at least one negative* Vote *message is collected by an application server from some database server or (2) at least one* Status *message with abort is collected by an application/database server from some database server. Note that this implies that either (i) the insertion of the ITP with an abort value is successful on at least one database, or (ii) the insertion of the ITP with the prepared value for the transaction state fails on at least one database, or is not attempted at all due to the fact that* `xa_prepare` *returns no at that database.*

3.3 Protocol Correctness

In this section we provide a formal proof of the protocol correctness with respect to the e-Transaction properties, as formally specified in [44, 45]. Since the protocol proposed in this work represents a solution to the very same problem, we inherit its specification, which is recalled below. As hinted, there are three categories of properties that define the e-Transaction problem: Termination, Agreement, and Validity. These properties are specified as follows:

Termination:

T.1 If the client issues a request, then, unless it crashes, the client eventually delivers a result.

T.2 If any database server votes for a result, then the database server eventually commits or aborts the result.

Agreement:

A.1 No result is delivered by the client unless the result is committed by all database servers.

A.2 No database server commits two different results.

A.3 No two database servers decide differently on the same result.

Validity:

V.1 If the client delivers a result, then the result must have been computed by an application server with, as a parameter, a request issued by the client.

V.2 No database server commits a result unless all database servers have voted *yes* for that result.

The intuitive meaning of the previous properties has been already pointed out in Chapter 2. As the only additional note, the above properties express guarantees on data integrity (e.g. distributed transaction atomicity - see **A.3**) and data availability (e.g. the ability of a database server not to maintain pre-committed data blocked forever - see **T.2**) independently of what happens to the client. This well fits the "pure" crash model for the client (i.e. the client is not required to recover after a failure), which allows the e-Transaction framework to deal with very thin clients not providing access to stable storage. (We again recall that access to stable storage might be precluded not only because of hardware constraints, but also due to security and privacy issues.)

We proceed by first introducing the assumptions required to prove the correctness of our protocol. Next, we prove the seven e-Transaction properties individually, by also relying on lemmas we introduce in order to simplify the structure of the proof.

3.3.1 Correctness Assumptions

We assume that at least one application server in the set $\{AS_1, \ldots, AS_n\}$ is correct, i.e. it does not crash. This assumption is required to ensure protocol termination since, in compliance with the e-Transaction framework, application servers adhere to a pure crash model. On the other hand, in practical settings, our protocol can still guarantee the e-Transaction properties even in the case of the simultaneous crash of all the application servers, as long as at least one of them eventually recovers and remains up long enough to complete the whole end-to-end interaction (application server recovery would not require any particular handling since application servers are assumed to be stateless processes).

Like in [43, 45], we assume that all database servers are good, which means:

(1) They always recover after crashes, and eventually stop crashing (i.e. eventually they become correct), and

(2) If the application servers keep re-trying transactions, these are eventually committed.

Note that assuming the databases recover and eventually stop crashing means in practice assuming that application data are eventually available long enough to allow the end-user to successfully complete its interaction with the system. In practical settings, this assumption can be typically supported via a set of solutions, especially relying on replication and redundancy (see, e.g., [3, 29, 93, 94, 129, 122, 107]). On the other hand, admitting the possibility for a database not to recover and remain up would lead to the extreme (and, on the basis of those solutions, not very realistic) case in which the whole application remains indefinitely unavailable. As a note, we remark that assuming database goodness is one point which differentiates the problem we are addressing here (i.e. e-Transaction) from the classical non-blocking atomic commit problem [15, 98], where transactional processes are assumed not to recover (and remain up) after a crash.

We assume that failure detection of clients and database servers against application servers is supported by a $\diamondsuit S$ (eventually strong) failure detector. Actually, this is a very weak failure detector class, whose properties can be expressed, according to its specification in [25] (and to the case of failure detection against application servers), as follows:

- *Strong Completeness.* Eventually every application server that crashes is permanently suspected by every correct process.

- *Eventual Weak Accuracy.* There is a time after which some correct application server is never suspected by any correct process.

Before proceeding we note that weaker assumptions than those introduced in this section can only lead to violations of Termination properties (i.e. liveness), but have no impact on Agreement and Validity properties (i.e. safety). As an example, the $\diamondsuit S$ failure detector is required in order to ensure that at least one application server will be eventually given enough time to complete transaction processing, thanks to the avoidance of premature activation of the fail-over phase through Terminate messages. This would not be guaranteed by a failure detector which does not ensure the Eventual Weak Accuracy property.

3.3.2 Correctness Proof

Lemma 3.3.1 *If a correct application server receives either a* Request *or a* Terminate *message from a client, it eventually sends the corresponding* Outcome *message to the client.*

Proof Suppose a correct application server receives either a Request or a Terminate message from a client. In this case, given that all the primitives available at the application server are non-blocking, the correct application server keeps on retransmitting, on a timeout basis, Prepare/Decide messages (case of Request from the client), or Resolve messages (case of Terminate from the client) to the database servers until Vote/Outcome or Status replies are received from all of them. Let t be the time after which all the database servers stop crashing and remain up. After t, by channel reliability, all the database servers eventually receive the above messages, triggering the activation of the corresponding database server tasks. Since these tasks execute only non-blocking primitives, all the database servers eventually send Vote/Outcome or Status messages to the correct application server. Given that channels are reliable, these messages are eventually received by this server. (A Status message might trigger a new round of interaction with the database servers through Decide messages which also eventually lead to a reply from the databases through Outcome messages, given that we are at time after t.) Hence the correct application server is able to eventually send back to the client the Outcome message. $Q.E.D.$

Lemma 3.3.2 *If the client issues a request, then unless it crashes, it eventually receives a corresponding* Outcome *message.*

Proof Consider a client that sends a Request message and that does not crash. There are two cases:

(1) The message is sent to a correct application server. By reliability of communication channels, this message is eventually received by this server that, by Lemma 3.3.1, eventually sends the Outcome message back. Again by reliability of communication channels, this message is eventually received by the client, since it does not crash. Hence the claim follows.

(2) The message is sent to a non-correct application server. Suppose by contradiction that the client does not get the Outcome message. In this case, by the completeness property of the failure detector, the client eventually suspects this server and sends a Terminate message to another application server. At this point we have two cases:

(2.1) The Terminate message is destined to a correct application server. In this case (by reliability of communication channels and Lemma 3.3.1) the client gets the Outcome message, hence the assumption is contradicted and the claim follows.

(2.2) The Terminate message is destined to a non-correct application server. In this case, since we have assumed that the client does not get the Outcome message, by the completeness property of the failure detector, the client eventually suspects this server and sends a Terminate message to another application server.

We note however that case 2.2 cannot occur indefinitely, since a Terminate message is eventually sent to a correct application server (recall we have assumed that there is at least one correct application server). Hence we eventually fall in case 2.1 and the claim follows.

Q.E.D.

Lemma 3.3.3 *If a database server suspects an application server, the database server eventually decides for every transaction already pre-committed by that application server.*

Proof If a database server, say DB_i, suspects an application server, say AS_i, this means that DB_i has pre-committed a transaction T initiated by AS_i, and has succeeded in inserting the ITP with *prepared* value. In this case DB_i starts sending Resolve messages for the transaction to all the database servers on a timeout basis (this occurs even if DB_i crashes and then recovers). Let t be the time after which all the database servers stop crashing and remain up. After t, by channel reliability, all the database servers eventually receive the above Resolve messages, triggering the activation of the corresponding database server tasks. Since these tasks execute only non-blocking primitives, all the database servers eventually send back to DB_i Status messages, which are eventually received. Next, DB_i broadcasts to all the database servers (including itself) a Decide message carrying either the *abort* or *commit* decision. By the same previous arguments (i.e. reliability of channels and non-blocking primitives) these messages are eventually received by all the database servers and processed since we are at time after t, which leads DB_i to eventually take a decision for the transaction T. Hence the claim follows. *Q.E.D.*

Lemma 3.3.4 *If a database is asked to decide abort or commit for an unknown transaction (this is the case of the unknown_tid in Task 2), the database has already taken the same decision it is currently asked to take.*

Proof By Observation 3.2.1 and Observation 3.2.2 in Section 3.2.4, no two application/database servers can take a different decision on the outcome of any distributed transaction, since the conditions enabling the send of a Decide message with the *commit* or *abort* indication are mutually exclusive. As a direct consequence, if a database server is asked to decide *commit* or *abort* for an unknown transaction, then it cannot have taken a different decision. Hence the claim follows. *Q.E.D.*

Termination T.1 - *If the client issues a request, then, unless it crashes, the client eventually delivers a result ([7]).*

Proof By Lemma 3.3.2, if the client issues a request and does not crash, it eventually receives an Outcome message. There are two cases:

(1) If the Outcome message carries the *commit* indication then the client delivers the result and the claim follows.

(2) If the Outcome message carries the *abort* indication, the client re-submits a new request and continues to do so until an Outcome message carrying the *commit* indication is received. However, case 2 cannot occur indefinitely since there will be a time t after which all the database servers stop crashing and remain up, and the retransmitted requests arrive to a correct application server (recall we have assumed there is at least one of such correct processes), which, by the accuracy property of failure detection, is never suspected by the client or by database servers. Hence neither does the client send Terminate messages nor do the databases send Resolve messages to attempt the abort of the transaction. Given that the database servers are good, the correct application server is able to eventually commit the transaction. Therefore, the Outcome message sent to the client eventually carries the *commit* indication. Hence, the claim follows.

Q.E.D.

Termination T.2 - *If any database server votes for a result, then the database server eventually commits or aborts the result.*

[7] In the rest of this section, we use "vote/decide for a result" as synonymous with "vote/decide for a transaction". Similarly, "commit/abort a result" is used as synonymous with "commit/abort a transaction". This is done in order to maintain the same terminology used in [43, 44, 45] for the presentation of e-Transaction properties while ensuring, within the discussions, compatibility with the presentation of our protocol. For this same reason, delivery of the result at the client side expresses that the `issue` method returns a result associated with a committed transaction.

Proof Suppose a database votes for a result (i.e. pre-commits the transaction) but fails to insert the corresponding ITP with *prepared* value. This is either because the database server crashes before the ITP insertion (in this case the transaction is aborted upon recovery) or because the `try_abort` method successfully inserts the ITP with *abort* (in this case the transaction is aborted by this method or upon recovery). In either case the claim follows.
On the other hand, if the ITP with the *prepared* state value is successfully inserted after the vote, we have two additional cases.

(1) The application server that pre-committed the transaction crashes. Then, by the completeness property of failure detection, the database server eventually suspects the application server and by Lemma 3.3.3 decides on the transaction. Hence the claim follows.

(2) The application server that pre-committed the transaction is correct. Then, by Lemma 3.3.1, this application server eventually sends an Outcome message to the client. This implies that the application server has ensured that all the database servers have decided for that transaction. Hence the claim follows.

Q.E.D.

Agreement A.1 - *No result is delivered by the client unless the result is committed by all database servers.*

Proof The client delivers the result only when an Outcome message with the *commit* indication is received from an application server. On the other hand, the application server sends the Outcome message to the client only after an Outcome message indicating *commit* is received by all database servers, which means that the result has been committed by all database servers. *Q.E.D.*

Agreement A.2 - *No database server commits two different results.*

Proof For a database server to commit two different results, we need the client to send at least two requests to the application servers. The client resubmits a new request only after it has received the Outcome message from an application server carrying the *abort* indication for the result associated with the last issued request. On the other hand, the application server returns to the client the Outcome message with the *abort* indication only after each database server either has already recorded an ITP with the *abort* state or has voted *no* for that result. As a consequence each time a new request is issued

by the client, no previous request can eventually be committed since after the *abort* state is logged within the ITP, the database server rejects voting *yes* for that result. The same happens in case the database server already voted *no* since the `xa_prepare` primitive does not recognize the identifier associated with the result. As a consequence, no two different results can eventually be committed by a database server. *Q.E.D.*

Agreement A.3 - *No two database servers decide differently on the same result.*

Proof A database can decide *commit* only if it receives a Decide message with the *commit* indication, whereas it can decide *abort* if either (1) it receives a Decide message with the *abort* indication, or (2) it receives a Resolve message that causes a successful insertion of the ITP with the *abort* value for that result. By Observation 3.2.1 and Observation 3.2.2 in Section 3.2.4, no two Decide messages with different indications (*commit* or *abort*) can ever be sent, since the conditions enabling the send of Decide messages with different indications are mutually exclusive. Hence no two database servers can ever decide differently due to case 1. On the other hand, after the insertion of the ITP with the *prepared* value in **Task 1**, the database server will reject any successive insertion of the ITP, thus avoiding the abort of the transaction due to the receipt of Resolve messages activating the execution of **Task 3**. Therefore, when a database server receives a Decide message to commit a transaction (by Observation 3.2.1 this implies that all the database servers have performed successful insertion of the ITP with the *prepared* value for the transaction state) it is sure that no database server will ever accept aborting that transaction due to Resolve messages. Hence no two database servers can ever decide differently due to case 2. *Q.E.D.*

Validity V.1 - *If the client delivers a result, then the result must have been computed by an application server with, as a parameter, a request issued by the client.*

Proof The client delivers the result after it receives an Outcome message with the *commit* indication from an application server for a given identifier. Such a message is sent to the client if, and only if, the application server has received from each database server an Outcome message with the *commit* indication for that identifier. This happens only if the result has already been committed. Given that a transaction associated with an identifier is committed only after an application server has computed and prepared it on all the databases, and

given that the application server computes and prepares the transaction only after it has received the Request message with that identifier from the client, then it is not possible that the client delivers the result unless it has issued a request that has been computed by the application server. *Q.E.D.*

Validity V.2 - *No database server commits a result unless all database servers have voted yes for that result.*

Proof A database server commits a result after it receives the Decide message for that result with a *commit* indication. On the other hand, by Observation 3.2.1 in Section 3.2.4, such a message can be sent only after every database server has logged an ITP for that result storing a *prepared* state value. By the database server pseudo-code, an ITP with *prepared* state value is logged only after the database server voted *yes* for that result. Thus, if a database server commits a result, then all database servers must have voted *yes* for that result. *Q.E.D.*

3.4 Garbage Collection and Other Practical issues

The following mechanism could be coupled with the protocol to deal with garbage collection of unneeded recovery information (i.e. unneeded ITPs) from the databases.

If the client experiences a nice run (i.e. a run with no suspect of failure), then an acknowledgment message can be sent to the application server right after the Outcome message has been received at the client side. Upon receipt of this message, the application server simply issues a request for discarding the corresponding ITPs to the back-end databases.

On the other hand, if the client receives the Outcome message for a given request identifier (possibly with the *abort* outcome) only after having suspected the application server and after having sent Terminate messages for that request, the ITP (possibly inserted at each back-end database with the *abort* value during the resolve phase handled by the application server) needs to be maintained. This is done to avoid that the transaction associated with that request identifier, which is asynchronously processed by the originally contacted application server, gets eventually committed, hence violating, e.g., the **A.2** e-Transaction property due to a retransmission of a different request instance by the client after the receipt of the *abort* outcome. In such a scenario, anyway, the client could send a different type of acknowledgment message to the application server. Upon receipt of this message the application server could inform the database servers to discard the transaction result (while maintaining the

other information) from the ITP entry associated with every transaction for which a Terminate message was sent from the client. This is done since the result (e.g., an HTML page) is expected to occupy most of the storage required for the ITP tuple. Hence discarding it means in practice freeing almost all the storage allocated for the recovery information associated with a given client request. We note however that, in practical life, nice runs represent the most common cases. Hence, discarding the ITP in nice runs, while maintaining such a recovery information only in case of unlikely failure (or suspect of failure) situations, means in practice very limited growth of storage usage for recovery information over time.

Garbage collection without acknowledgments from the client could be further addressed if it were possible to determine a known period of time after which the client does not retransmit requests and the application/database servers do not attempt to perform further processing/resolve actions for a given request. As already discussed in [44], these mechanisms are feasible if the underlying system matches assumptions proper of a timed model, e.g. [32]. Anyway, in practical settings one could still rely on approaches based on the association of an adequately selected Time-To-Leave (TTL) with each ITP, after which deletion of the ITP itself from the database can be performed.

As an additional note, we have presented the protocol under the assumption that all the back-end databases are accessed by the distributed transaction. However, for real life applications on, e.g., large scale systems, it could be possible that a request from a client needs transactional access only to a subset of those databases. Dealing with such a case would require a mechanism for persistently associating the client request with the identities of the subset of back-end databases really involved in the distributed transaction. In case the association can be deterministically defined by the application server as a function of the client request content, persistence of the association could be supported by including within the ITP the identities of the set of involved database servers, so that they could be able to execute the resolve phase associated with pending pre-committed transactions at the back-end tier (see **Task 4** in Section 3.2.3). The request content should also be piggybacked on Terminate messages from the client so that each application server can determine the set of involved database servers if the resolve phase is handled by the middle-tier.

On the other hand, if the association between the request content and the accessed back-end databases cannot be deterministically defined, one could rely on the following mechanism. The client request identifier can be used as the input of, e.g., a hash function determining the identity of a given database, say DB_x, on which the insertion of the ITP is forced during the prepare/resolve phase, even though that database would not be required to be accessed by the application logic. At the same time, the identities of the databases really

involved in the transaction, plus the identity of DB_x, can be logged within the ITP. In this way, similarly to the above case, a database server keeping a pre-committed transaction is able to activate the resolve phase since the list of the involved databases is available within the ITP kept for that transaction. Also, an application server receiving the Terminate message from the client should send a Resolve message to DB_x (identified via the hash function) for either (i) retrieving the list of the involved databases, if the ITP with the *prepared* state value has been inserted at DB_x, in order to execute the resolve phase on all of them, or (ii) allowing eventual abort of that transaction by performing the insertion of the ITP with the *abort* value on DB_x ([8]).

Overall, in the case of deterministically defined association between the client request content and the set of involved back-end databases, we would only require slightly larger storage space for the ITP, but no additional log operations. On the other hand, in the case of non-deterministically defined association between the client request content and the set of involved back-end databases, we might require an additional round of messages in between the application server and DB_x during the resolve phase, in order to retrieve the list of the other database servers involved in the transaction. However, this additional communication cost does not need to be paid in nice runs. At the same time, a properly defined hash function allowing even distribution of additional forced ITP insertions (i.e. additional load) across the whole back-end tier would also prevent bottlenecks.

Finally, our protocol has been presented for the case of clients not recovering after a crash. This has been done in compliance with the e-Transaction specification, which, as already hinted, assumes a pure crash model for the clients. However, if stable storage capabilities were allowed at the client side, our protocol could be easily extended to deal with clients recovering after a crash. This could be done by logging Request/Outcome messages, and by having the client send out, upon recovery, Terminate messages for any Request message not having the corresponding Outcome in the log.

3.5 Performance Measures and Comparison

In this section, we aim at quantitatively comparing the performance of our protocol against existing solutions. We carry out the comparison by presenting simple, yet realistic, models for the response time of each protocol, and studying the output provided by the models while varying some system parameters. We are interested in the case of no data contention and light system

[8] In case the transaction has been prepared at some database other than DB_x, this database will eventually abort the transaction during a resolve phase since this phase will find out the *abort* indication within the ITP maintained by DB_x.

load. This allows us to evaluate the impact of each protocol on response time more accurately, since we avoid any interference due to overhead factors not directly related to the distributed management of transaction processing performed by the protocols themselves (e.g. the overhead caused by delay in the access to data within the databases due to contention).

We are interested in studying the case of normal execution (nice runs), i.e. when no process crashes or is suspected to have crashed. This is because, as already hinted in Section 3.4, those runs are the most likely to occur in practice, thus representing an adequate test-bed for the evaluation of the real performance effectiveness of any solution. Anyway, it is worth remarking that our protocol supports in practice the transfer of the transaction coordinator role over the middle-tier without explicit coordination among different application servers (transfer is triggered by the client retransmission logic). Hence, as soon as there is at least one available (i.e. up and working) application server instance, system availability when our protocol is employed only depends on the availability of back-end database servers. As already hinted in Section 3.3.1, this can be typically guaranteed via a set of solutions (see, e.g., [3, 29, 93, 94, 129, 122, 107]).

For simplicity of presentation, but with no loss of generality and without penalizing any of the compared protocols, we will focus on response time as seen by the application server, thus omitting the round-trip time between client and application server. We compare our protocol with the following alternatives, all addressing the scenario of atomic transactions spanning multiple databases ([9]):

1. A baseline protocol that coordinates distributed transaction processing via 2PC without logs on the coordinator (see Figure 3.6.a). This protocol tolerates crashes, with recovery of the back-end databases only.

2. The persistent queue (PQ) approach [16], whose behavior is schematized in Figure 3.6.b. This approach performs the enqueue of the client request as the first action. Next, it requires START and PRECOMMIT logs at the application server, i.e. classical 2PC, to guarantee atomicity of the distributed transaction. As already mentioned, this transaction also includes the enqueuing of the result of the data manipulation performed during the compute phase.

3. The primary-backup replication scheme (PBR) presented in [45] and the asynchronous replication scheme (AR) presented in [44]. The behavior of both these protocols can be schematized as shown in Figure

[9] The e-Transaction protocol in [43] is excluded from this comparison as it tackles the more restricted case of single database in the back-end tier.

3.6.c, where the COORDINATION phase represents either the activity of propagating recovery information (i.e. the client request and the transaction result) from the primary application server to the backups (this holds for PBR) or the activity of updating the consensus object, i.e. the write-once register (this holds for AR).

For completeness, we also show (see Figure 3.6.d) the schematized behavior of our protocol. Compared to the baseline we simply add (i) the insertion of the ITP with *prepared* state value (this is done before the database server sends out the Vote message) and (ii) the update of the state maintained by the ITP to the *commit* value. However, the latter action is performed after the database server sends out the Outcome message to the application server. Therefore the cost of this operation does not contribute to the response time perceived by the application server.

While building the response time models, we suppose with no loss of generality that (i) the back-end databases have the same computational capacity and that (ii) the round-trip time between the application server and each back-end database $RTT_{as/db}$ is equal for all the databases. (Actually, in the case of heterogeneous databases and/or different round-trip times with the application server, the expressions we propose are still representative when considering the maximum value, across all the databases, for the terms they contain.) Also, we model the case of transactional logic activated via a single message, e.g. like in stored procedures, in order to avoid the introduction of an arbitrary delay in the response time models caused by an arbitrary number of message exchanges between application and database servers for the management of the transactional logic.

Concerning the response time of the baseline protocol $T_{baseline}$ in the case of normal execution, seen by the application server, we have the following expression:

$$T_{baseline} = (T_{SQL} + T_{xa\_prepare} + T_{xa\_commit}) + 3 \times RTT_{as/db} \tag{3.1}$$

where the term $RTT_{as/db}$ takes into account the fact that activation of functions in the XA interface and of the SQL associated with the transaction needs a message exchange from the application server to the database server and the corresponding acknowledgment.

All the other protocols execute the same actions of the baseline plus additional actions (see Figure 3.6). Hence, the response time of each protocol can be expressed as follows:

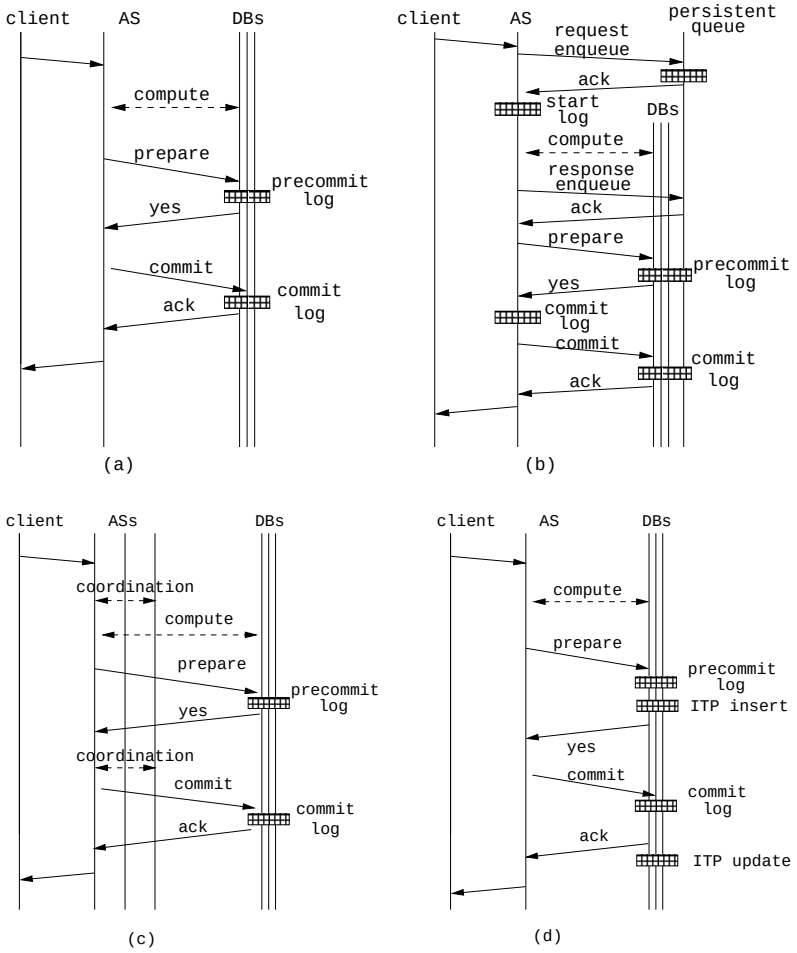


Figure 3.6: Schematization of the (Normal) Behavior of the Compared Protocols. (Filled boxes represent eager disk accesses)

common parameters			PQ			
T_{SQL}	$T_{xa\_prepare}$	T_{xa_decide}	T_{start}	$T_{precommit}$	$T_{req-enqueue}$	$T_{res-enqueue}$
186.78	6.07	9.99	1.66	0.44	20.30	0.77

our protocol	
T_{ITP_insert}	T_{ITP_update}
2.22 (file system) or 20.30 (SQL)	0.46 (file system) or 0.81 (SQL)

Table 3.1: Measured Parameters Values (Expressed in ms).

$$T_{PQ} = T_{baseline} + T_{start} + T_{precommit} + 2 \times RTT_{as/qs} + \\ + T_{req-enqueue} + T_{res-enqueue} \quad (3.2)$$

$$T_{PBR} = T_{AR} = T_{baseline} + 2 \times T_{coordination} \quad (3.3)$$

$$T_{our\_protocol} = T_{baseline} + T_{ITP\_insert} \quad (3.4)$$

where $RTT_{as/qs}$ is the round-trip time between an application server and the queuing system used by PQ, and $T_{req-enqueue}$ (resp. $T_{res-enqueue}$) represents the time to enqueue the client request (resp. the transaction result) within that system.

Some parameters appearing in the latency models are left as independent variables in the performance study. They are:

(i) $RTT_{as/db}$, typically dependent on the relative locations of application servers and database servers;

(ii) $RTT_{as/qs}$, typically dependent on the relative locations of application servers and the persistent queuing system; and

(iii) $T_{coordination}$, which depends on the specific algorithm selected for either the management of the update of backup application servers in PBR, or the management of the consensus object (i.e. the write-once register) in AR, and also on the speed of communication among application server replicas.

Other parameters have been measured through prototype implementations of PQ and of our protocol, relying on DB2 V8.1 and LINUX (kernel version 2.4.21). Our prototype of PQ performs the START and PRECOMMIT logs via operations on the file system, as in the approach commonly used by transaction monitors [17]. Also, as typically found in industrial environment (e.g. [113]), persistent message storing is supported through a database system,

namely DB2 in our case. For what concerns our protocol, we have developed two different implementations. The first one manipulates the ITP via local transactions on DB2. In this case the ITP is maintained inside a user level database table. The second implementation is instead based on an optimized approach relying on the LINUX file system. In this case the ITP is maintained on files and ACID properties, as well as support for primary key constraint semantic, are ensured via standard file locking and synchronous operations on the disk. In order to use a representative value for T_{SQL} in the comparative study, we have also implemented the TPC BENCHMARK^{TM} C (New-Order-Transaction) [119] (this benchmark portrays the activity of a wholesale supplier), and measured the latency for the related SQL operations. Table 3.5 lists obtained measures for the case of application server and database server both hosted by a Pentium IV 2.66GHz with 512MB RAM and a single UDMA100 disk. The two different values for T_{ITP_insert} and T_{ITP_update} refer to the cases of ITP managed through either the file system or local transactions on DB2. (As already pointed out, T_{ITP_update} does not contribute to the response time of our protocol as seen by the application server side since the ITP update is executed after the database server has sent the outcome to the application server. However we report the obtained measure for this parameter in order to provide the reader with indications on the update cost at the database side.) Each reported value, expressed in msec, is the average over a number of samples that ensures a confidence interval of 10% around the mean at the 95% confidence level. As we are interested in the case of no data contention and light system load, all the measures have been taken for the case of requests submitted one at a time.

We provide now quantitative comparisons among the protocols based on the proposed response time models and the obtained measurements. We analyze two different, representative scenarios. In the first one, application servers and database servers are assumed to be located on the same LAN; the same happens for the persistent queuing system used by PQ. In this case we have set $T_{coordination} = RTT_{as/qs} = RTT_{as/db}$ ([10]) and have varied the value of these parameters between 50 μsec and 10 msec so as to figure out different settings for what concerns the speed of the network and the communication layer among application servers, or among application servers and the queuing system in PQ. The case of 50 μsec is representative of settings in which the

[10] In the absence of faults, a round of messages is the lower bound on message complexity required for transmission and acknowledgment of recovery information between the primary and the backups in the PBR solution. Also, as shown in [63], in the absence of faults a round of messages is the lower bound on message complexity for achieving consensus, i.e. for the management of the consensus object in AR. Therefore, the lower bound on the latency of coordination can be modeled as a round trip among two application servers, which is equal to $RTT_{as/db}$ if all the processes are hosted by machines on the same LAN.

hosts on the LAN are connected, e.g., by a high speed switch, coupled with an ad-hoc message passing layer [91]. Instead, the case of 10 msec might be representative of a slower LAN with less performance effective message passing (e.g. based on RPC stacks such as RMI). In the second scenario we analyze, database servers are assumed to be spread on the Internet, thus $RTT_{as/db}$ has been set to the reasonable value of 250 msec [59]. Instead, $T_{coordination}$ and $RTT_{as/qs}$ have been varied between 1 and 250 msec so as to capture different organizations with respect to the spatial location of application server replicas and the queuing system in PQ. Lower values capture the cases in which those replicas (and the queuing system) are distributed, e.g., over the same LAN. The extreme value of 250 msec captures, instead, the opposite case in which they are distributed, e.g., over the Internet.

Figure 3.7 shows the percentage of overhead of the protocols compared to the baseline. For the first scenario, we observe the following tendencies. The optimized implementation of our protocol based on file system API exhibits a negligible overhead percentage. For minimal values of $T_{coordination}$ and $RTT_{as/db}$, i.e. up to 0.5 msec, PBR and AR show the lowest overhead. On the other hand, as soon as the round trip time on the LAN hosting the processes gets larger, these protocols tend to perform similarly to the less efficient implementation of our protocol based on SQL. As an extreme, for round trip time of 10 msec, they show an increase in the overhead of up to 9 times as compared to the optimized implementation of our protocol. On the other hand, PQ performs worse than our protocol independently of the selected values for $RTT_{as/qs}$ and $RTT_{as/db}$, with gain from our protocol that increases while the values of $RTT_{as/qs}$ and $RTT_{as/db}$ increase. We note however that, independently of the relative values of their overhead percentages, all the protocols, except PQ, actually exhibit additional overhead compared to the baseline which is at most of 10%. With respect to the latter point, the optimized implementation of our protocol shows overhead percentage constantly under 1.5%. In the second scenario our protocol outperforms PBR, AR and PQ, for almost any considered value of $T_{coordination}$ and $RTT_{as/qs}$. Specifically, these protocols show overhead percentage comparable to our protocol only if $T_{coordination}$ (resp. $RTT_{as/qs}$) is maintained up to 10 msec (resp. 1 msec). On the other hand, as soon as these parameters assume a larger value, the overhead percentage of these protocols definitely grows. As an extreme, with the value of 250 msec, such an overhead percentage gets up to 27 times larger than the overhead percentage of the SQL based implementation of our protocol. Also, differently from the first scenario, this time PBR, AR and PQ exhibit a remarkable additional overhead compared to the baseline, i.e. up to 55%. Instead, our protocol keeps the additional overhead over the baseline constantly under 2% for the SQL based implementation, and under 0.3% for the optimized implementation relying on file system API. Overall, our protocol

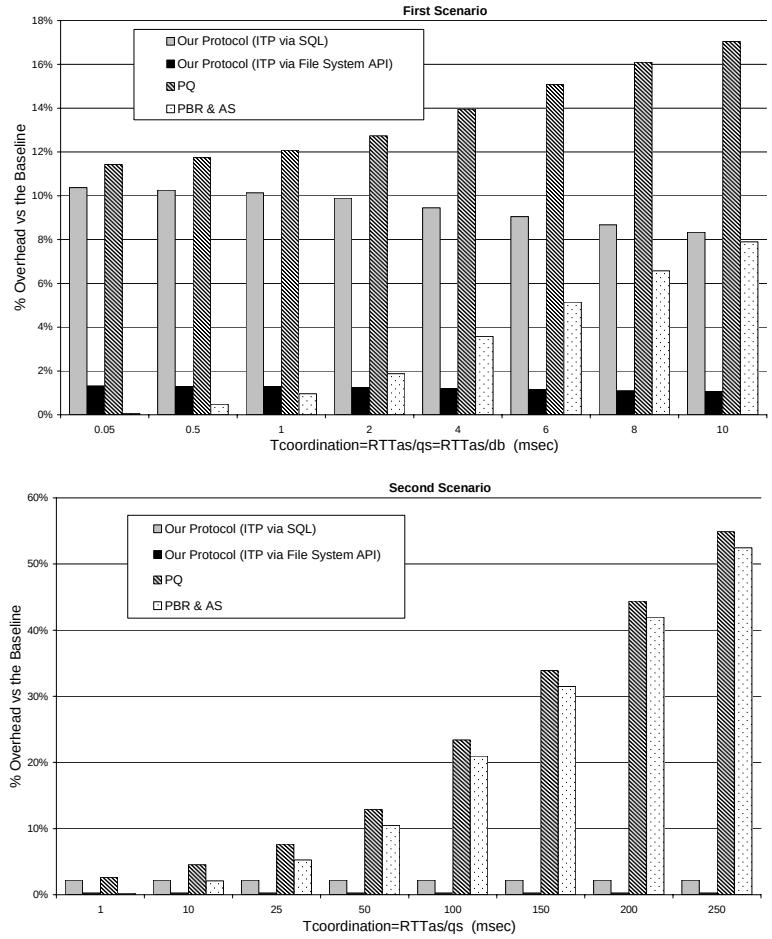


Figure 3.7: Overhead Percentage vs the Baseline Protocol.

keeps the percentage of overhead over the baseline under 10% (this is drastically reduced to 2% in case of the optimized implementation) for whichever considered settings of both the investigated scenarios. This points out how, differently from other proposals, it exhibits good performance independently of the particular system organization, thus being attractive for usage in both the cases of local and wide area distribution of the application/database servers.

Chapter 4

Ensuring e-Transaction with no Assumption on the Accuracy of Failure Detection

In the previous chapter we have introduced an e-Transaction protocol which, compared to state of the art approaches, avoids any form of explicit coordination among middle-tier application servers, while providing these servers with provide mutual fail-over capabilities. The correctness of that protocol relies on the ability of the employed failure detection mechanism to provide an adequate (although relatively weak) level of accuracy. (We recall that the accuracy property of a failure detector embodies its ability not to indefinitely falsely suspect correct processes to have crashed.) In this chapter we present an e-Transaction protocol that, beyond the avoidance of application server coordination schemes, exhibits the distinguishing features of being suited for asynchronous systems and of not relying on any assumption on the accuracy of failure detection. This allows a further enlargement of the spectrum of system organizations covered by the provided solution, including general Web infrastructures where inaccurate fault monitoring and detection cannot be avoided, like in the case of infrastructures belonging to (and controlled by) providers offering different levels of guarantees, or even no guarantee at all, on, e.g., the processing speed or the message transmission delay.

We note that the lack of accuracy in the failure detection may lead to the pathological situation in which false failure suspicions are issued indefinitely while handling the end-to-end interaction. In such a scenario, an extermination based approach before re-issuing requests (like the one adopted by the protocol we have presented in Chapter 3 via the termination/resolve phase) might yield to an indefinite sequence of aborts of on-going work carried out on behalf of a given client by falsely suspected servers. On the other hand,

if no extermination is performed, re-issuing requests might lead to blocking situations (due to durable pre-commit locks) involving both newly activated and previously activated work carried out by falsely suspected servers. In both cases, liveness can get compromised.

To overcome these problems, the protocol we provide in this chapter exploits an innovative scheme for distributed transaction management, based on ad-hoc demarcation and concurrency control mechanisms, which we refer to as Multi-Instance-Precommit (MIP). With this scheme, we allow a falsely suspected server to proceed with transaction processing and pre-commit (i.e. no attempt to force the abort of its work is performed). Also, any server performing fail-over of a client request is granted access to the pre-image of any uncommitted data item updated by (falsely) suspected servers previously processing that same client request. In this way newly activated work in case of fail-over does not need to force the abort of previously activated one, and the two works do not block each other, which provides liveness guarantees on the end-to-end interaction. At the same time, the different (pre-committed) work instances are reconciled at commit time to maintain application safety (e.g. at-most once semantic for request processing).

Given that the MIP scheme is custom, and is actually not supported by current, conventional database technology, this protocol cannot be straightforwardly implemented on top of conventional systems, which instead can be done for the protocol presented in Chapter 3 (this expresses a trade-off between the two provided solutions). However, in this chapter we also discuss hints on how to integrate the MIP scheme, as well as the whole e-Transaction protocol, with conventional software systems.

4.1 Model of the System

Concerning client and application server processes, we consider here the same identical model introduced in Section 3.1 of Chapter 3. Hence we focus on the discussion of the database server model, which allow us to introduce the MIP facility used as the building block for the construction of the e-Transaction protocol.

4.1.1 Database Servers

Back-end database servers support distributed transactions according to the innovative MIP scheme, whose features are described below:

Transaction Demarcation

Database servers associate with each transaction an identifier, namely a XID, which is composed by (1) a request identifier, namely *req_id*, univocally associated with a given client request, and (2) a transaction instance identifier, namely *inst_id*, composed of a tuple $< category, instance\_number >$, where:

- *category* identifies a given process (client or database server);
- *instance_number* is a numerical value greater than or equal to zero.

We assume that category values are ordered according to a lexicographic relation such that $category < category'$ if *category* identifies a client process and $category'$ identifies a database server. Also, in case both the values identify two database servers, we say that $category < category'$ in case *category* identifies a database server which precedes the one identified by $category'$ when ordering the set $\{DB_1, \ldots, DB_m\}$ according to some predetermined scheme (i.e. the set is ordered on the basis of database servers indexes).

Exploiting the previous ordering relation on category values, we assume that *inst_id* values are ordered according to the following relation: $inst\_id =< category, instance\_number >$ is less than $inst\_id' =< category', instance\_number' >$ if (i) $instance\_number < instance\_number'$ or (ii) $instance\_number = instance\_number'$ and $category < category'$. In the following, transactions sharing the same the request identifier (*req_id*) value but having different transaction instance identifier (*inst_id*) values will also be referred to as *sibling transactions*.

Atomic Commit Protocol (ACP) Supports

We model with the primitives `prepare` and `decide`, the database server interface for supporting the ACP. The primitive `prepare` takes in input a XID (i.e. a request identifier and a transaction instance identifier) and returns a value in the domain $\{prepared, abort\}$ reflecting whether the database server is able to commit the transaction or not. The primitive `decide` takes in input a XID and a decision in the domain $\{commit, abort\}$, and commits or aborts that transaction (i.e. determines the final outcome for that transaction). This primitive commits a transaction only if it was already pre-committed and the input decision is *commit*. We assume that, if invoked with the *commit* indication for a prepared (i.e. pre-committed) transaction $XID =< req\_id, inst\_id >$, `decide` also aborts any pre-committed transaction with the same *req_id* having a transaction instance identifier $inst\_id'$ different from *inst_id*. With no loss of generality we assume both `prepare` and `decide` to be non-blocking.

Concurrency Control

In case a transaction T requires (read/write) access to some data item d previously accessed (written/read) by a not yet committed (e.g. pre-committed) transaction T', T is granted access to the pre-image of d with respect to the execution of T' if T and T' share the same req_id (i.e. they are sibling transactions). Hence any update performed by a not yet committed transaction T' is not visible to any sibling transaction T. On the other hand, no assumption is made on how concurrency control regulates data accesses of non-sibling transactions.

Multi Instance Pre-commit Tables

A Multi Instance Pre-commit table (MIPT) is persistently maintained by a database server for each set of transactions having the same req_id (i.e. sibling transactions originated by the same client request). In the following, we will denote with MIPT_x the table keeping track of transactions with $req\_id = x$. The y-th entry of MIPT_x, namely $\text{MIPT}_x[y]$, stores the following information related to the transaction with $req\_id = x$ and transaction instance identifier $inst\_id = y$:

(1) *state*: a value, in the domain $\{null, prepared, abort\}$, reflecting the transaction current state at that database (we assume that *null* is the default initialization value);

(2) *result*: the (non-deterministic) output produced by the execution of the transaction at the back-end databases (also in this case we assume that *null* is the default initialization value).

Each MIPT_x also keeps a special field, namely $\text{MIPT}_x.req$ which records the client request content that gave rise to the transactions with $req\_id = x$.

4.2 The Protocol

In this section we introduce the pseudo-code for the behavior of the different processes involved in the application, i.e. client, application sever and database server. For presentation simplicity this time we start with the database server behavior.

4.2.1 Database Server Behavior

The pseudo-code for the behavior of the database server within our e-Transaction protocol is shown in Figures 4.1 and 4.2. As in Chapter 3, we

```
Class DatabaseServer{
 List ASlist = {AS_1, ..., AS_n};
 ApplicationServer AS;
 TypeMIPT MIPT;
 Request req;
 State state;
 InstanceIdentifier inst_id;
 Outcome outcome;
 on stable storage Counter counter = InitialValue;

 void main(){
   while(true){
     cobegin
       || wait receive Prepare[req, < req_id, inst_id >, result] from AS_i // Task 1
          MIPT=vote(req, < req_id, inst_id >, result);
          send Vote[req_id, MIPT] to AS_i;
       || wait receive Decide[< req_id, inst_id >, decision] from AS_i// Task 2
          decide(< req_id, inst_id >,decision);
          send DecideACK[< req_id, inst_id >] to AS_i;
       || wait receive Resolve[< req_id, inst_id >] from AS_i// Task 3
          MIPT=resolve(< req_id, inst_id >);
          send Vote[req_id, MIPT] to AS_i;
       || background: // Task 4
          for every transaction < req_id, − > pre-committed longer than TIMEOUT period {
            req = MIPT_req_id.req;
            AS = ASlist.next();
            inst_id =< GetMyCategory(), + + counter >;
            send Request[req, < req_id, inst_id >] to AS;
            reset TIMEOUT period for that transaction;
          } // end for every
   } // end while
 } // end main

 TypeMIPT vote(Request req, XID < req_id, inst_id >, Result result){ ...}

 TypeMIPT resolve(XID < req_id, inst_id >){ ... }

} // end class
```

Figure 4.1: Database Server Behavior, Part 1.

```
TypeMIPT DatabaseServer::vote(Request req, XID < req_id, inst_id >, Result result){
  atomically do{
    if (MIPT_req_id does not exist) {create MIPT_req_id; MIPT_req_id.req = req;}
    if (MIPT_req_id[inst_id].state == null) {
      state = prepare(req_id, inst_id);
      if (state == prepared)
        MIPT_req_id[inst_id].(state, result) = (prepared, result);
      else
        MIPT_req_id[inst_id].(state, result) = (abort, null);
    }// end if
  } // end of atomic statement
  return MIPT_req_id;
} // end vote

TypeMIPT DatabaseServer::resolve(XID < req_id, inst_id >){
InstanceIdentifier x;
  atomically do{
    ∀x < inst_id do {
      if ( MIPT_req_id[x].state == null) MIPT_req_id[x].state = abort;
    }
  }//end of atomic statement
  return MIPT_req_id;
}//end resolve
```

Figure 4.2: Database Server Behavior, Part 2.

avoid describing the database server behavior during the transaction execution phase, which is abstracted through the application server's `compute` primitive. The database server executes three tasks triggered by the receipt of different types of messages, and an additional background task.

Task 1: Upon the arrival of the Prepare$[req, < req_id, inst_id >, result]$ message from an application server, the `vote` method is invoked, which atomically performs the following operations. If MIPT$_{req_id}$ does not exist (i.e. the database server is attempting to prepare a transaction associated with a given req_id for the first time), the database server creates it and stores the request content within it. In case the entry of MIPT$_{req_id}$ with index $inst\_id$ has a *null* state value (this always holds in case MIPT$_{req_id}$ did not exist and has been just created), the database attempts to prepare the transaction with $XID =< req\_id, inst\_id >$ by invoking `prepare`. In case the transaction is successfully prepared, the entry MIPT$_{req_id}[inst_id]$ is updated to store the *prepared* state value and the *result* specified by the Prepare message. Otherwise, MIPT$_{req_id}[inst_id]$ is updated with the *abort* value. Finally, when the `vote` method returns, MIPT$_{req_id}$ is sent back to the application server via a Vote message.

Task 2: Upon the arrival of the Decide$[< req_id, inst_id >, decision]$ message

from an application server, the `decide` primitive is invoked to determine the requested outcome (*commit* or *abort*) for the transaction. If the requested outcome is *commit*, `decide` (according to its specification) also enforces the abort of any other pre-committed transaction having the same *req_id*. Finally, a DecisionACK message is sent back to the application server.

Task 3: Upon the arrival of the Resolve[$< req\_id, inst\_id >$] message from an application server, the `resolve` method is invoked, which atomically performs the following operations. For all the values of x less than *inst_id*, it checks whether $\text{MIPT}_{req\_id}[x]$ has a *null* value. In the positive case, that value is set to *abort*. Finally, when the `resolve` method returns, MIPT_{req_id} is sent back to the application server via a Vote message.

Task 4: This is a background task used to avoid maintaining any pre-committed transaction blocked indefinitely. Within this task, the database server periodically checks whether there are transactions that are maintained in the pre-commit state longer than a timeout period. For each of these transactions, the original request content *req* is retrieved from the corresponding MIPT. Then, that same request is re-sent by the database server to whichever application server via a Request message. This message is also tagged with the original request identifier (i.e. *req_id* in the pseudo-code) and with a transaction instance identifier *inst_id* obtained by using (i) the database server identity as the category (i.e. the value $GetMyCategory()$) and (ii) an incremented counter value. Note that the used counter is assumed to be maintained on stable storage, which allows the database server to ensure monotonic increase of the counter even in case of recovery after a crash.

4.2.2 Observations

Observation 4.2.1 *By the database server pseudo-code, an update on whichever* $MIPT_{req\_id}[inst\_id]$ *entry can only occur once, i.e. when that entry has the null initialization value.*

Observation 4.2.2 *By the database server pseudo-code, whichever transaction* $XID =< req\_id, inst\_id >$ *can ever be prepared only if the corresponding entry* $MIPT_{req\_id}[inst\_id]$ *still keeps the null initialization value.*

4.2.3 Client and Application Server Behaviors

Figure 4.3 shows the pseudo-code defining the client behavior. Within the method `issue`, the client selects an application server and sends the request

to this server, together with the request identifier *req_id* (we abstract over the details for the determination of the request identifier via `SetId()`) and the transaction instance identifier *inst_id* formed by a category value identifying the client process and a counter value (i.e. the instance number) maintained by the client application. It then waits for the reply, namely for an Outcome message for the transaction ([1]). In case the Outcome message arrives, carrying the *commit* indication, `issue` simply returns the result of the transaction. On the other hand, if the outcome is *abort*, the client retransmits the same request after having incremented by one the counter used to define the transaction instance identifier. Otherwise, in case the contacted application server does not respond within a timeout period, the client selects a different application server and retransmits its request to this server, also in this case after having incremented by one the counter used to define the transaction instance identifier. Then it waits again for an Outcome message from an application server or for a timeout expiration.

```
Class Client{
  List ASlist = {AS_1, ..., AS_n};
  ApplicationServer AS;
  Result result;
  Outcome outcome;
  RequestIdentifier req_id;
  InstanceIdentifier inst_id;
  Counter counter = InitialValue;

   Result issue(Request req){
    outcome = abort;
    req_id = SetId(req);
    while(outcome == abort){
      AS = ASlist.next();
      set TIMEOUT;
      inst_id =< GetMyCategory(), ++counter >;
      send Request[req, < req_id, inst_id >] to AS;
      wait ( receive Outcome[< req_id, − >, outcome, result] from any AS_i ∈ ASlist
        or TIMEOUT );
    } // end while
    return result;
   } // end issue

  } // end class
```

Figure 4.3: Client Behavior.

The pseudo-code defining the behavior of an application server is shown in

[1] The notation $< req\_id, - >$ means that the instance identifier is a don't care value. Hence, the client actually waits for an Outcome message associated with the specified *req_id* and with whichever *inst_id* value.

Figure 4.4. The application server waits for a Request message from either a client or a database server. In case the Request message comes from a client, it is associated with either the original request transmitted by the client, or a request retransmission performed by the client. In case the request comes from a database server, say DB_i, it means that there is at least one pre-committed transaction instance associated with that same request, which has remained in the pre-commit state at DB_i for more than a timeout period (see Task 4 of the database server pseudo-code in Figure 4.1). In both cases, the Request message carries the request identifier (req_id) univocally associated with that client request, and the transaction instance identifier ($inst\_id$) defined by the identity of the sending process (client or database server) and by a monotonically increasing counter value. This simple scheme is sufficient to ensure that each Request message is univocally associated with a globally unique XID.

After the receipt of the Request message, the application server performs the compute phase for the corresponding transaction and determines the result set on each database. Then, it activates the first phase of the ACP protocol, during which it retransmits Prepare message to all the database servers on a timeout basis, until a Vote message is received from all of them. In our protocol, a Vote message from DB_i carries the $\text{MIPT}^i_{req\_id}$ maintained by DB_i for transactions associated with that req_id value. Therefore, at the end of the Vote collection phase, an application server is informed not only about the state of the transaction it is currently handling (i.e. the one whose XID it specified in the Prepare message), but also about the state of any sibling transaction at all the database servers.

Once collected $\text{MIPT}^i_{req\_id}$ from each database server DB_i, the application server verifies whether it is currently possible to take a positive (i.e. *commit*) decision for one of those sibling transactions. Specifically, the application server checks whether there is a transaction instance identifier j associated with req_id for which the following **Commit Condition** (**CC**) holds:

Sub-Commit-Condition-1 (SCC1): The transaction $XID =< req\_id, j >$ has been prepared at all the database servers (i.e. the condition $\forall i \in [1, m], MIPT^i_{req\_id}[j].state == prepared$ is verified); and

Sub-Commit-Condition-2 (SCC2): No sibling transaction having $XID =< req\_id, j' >$, with $j' < j$, may ever become prepared at all the database servers (i.e. the condition $\forall j' < j, \exists i \in [1, m] : MIPT^i_{req\_id}[j'].state == abort$ is verified - by Observation 4.2.2, if the database server keeps the value *abort* on a MIPT entry, the corresponding transaction instance cannot be ever prepared at that database).

If no transaction instance associated with req_id has been found prepared

```
Class ApplicationServer{
 List DBlist = {DB_1, ..., DB_m}; Outcome outcome; Result result;
 InstanceIdentifier PreparedInstance, CommittedInstance, InstanceToDecide;

 void main(){
  while(true){
    wait receive Request[req, < req_id, inst_id >] from client or from DB_i;
    result =compute(req, < req_id, inst_id >);
    repeat { // ACP vote phase
      send Prepare[req, < req_id, inst_id >, result] to each DB_i ∈ DBlist;
      wait (receive Vote[req_id,MIPT^i_req_id] from each DB_i ∈ DBlist) or TIMEOUT
    } until (received Vote[req_id,MIPT^i_req_id] from each DB_i ∈ DBlist);
    repeat{
      PreparedInstance = min(S), where S = {j : ∀i ∈ [1,m]  MIPT^i_req_id[j].state == prepared};
      if (PreparedInstance is not defined)
            { InstanceToDecide = inst_id; outcome = abort; break; }
      if (PreparedInstance is defined) and
        (∀j < PreparedInstance, ∃i ∈ [1,m] : MIPT^i_req_id[j].state == abort)
          { InstanceToDecide = PreparedInstance; outcome = commit; break; }
      repeat { // ACP resolve phase
        send Resolve[< req_id, PreparedInstance >] to each DB_i ∈ DBlist;
        wait (receive Vote[req_id,MIPT^i_req_id] from each DB_i ∈ DBlist) or TIMEOUT
      } until (received Vote[req_id,MIPT^i_req_id] from each DB_i ∈ DBlist);
    }until (TRUE);
    repeat{ // ACP decision phase
      send Decide[< req_id, InstanceToDecide >, outcome] to each DB_i ∈ DBlist;
      wait (receive DecideACK[req_id] from each DB_i ∈ DBlist) or TIMEOUT
    } until (received DecideACK[req_id] from each DB_i ∈ DBlist);
    if (Request was received from client){
      if (outcome == commit)
        set result to whichever received MIPT^i_req_id[InstanceToDecide].result;
      else result = null;
      send Outcome[< req_id, − >, outcome, result] to client
    }
  } // end while true
 } // end main
} // end class
```

Figure 4.4: Application Server Behavior.

at all databases (i.e. **SCC1** does not hold for any instance identifier, hence $PreparedInstance$ is not defined), then the application server makes sure that the transaction instance it is currently managing (i.e. the one associated with the received $inst\_id$) gets aborted at all the back-end databases. This is done by setting $InstanceToDecide$ to the value $inst\_id$, $outcome$ to the value $abort$, and then sending Decide messages with the negative (i.e. $abort$) indication for the transaction $XID =< req\_id, InstanceToDecide >$. These messages are re-sent on a timeout basis until acknowledgments are received from all the database servers.

If both **SCC1** and **SCC2** are verified (i.e. $PreparedInstance$ is defined and no other transaction associated with req_id and having instance identifier less than $PreparedInstance$ can eventually become prepared), the application server sets $InstanceToDecide$ to the value $PreparedInstance$, $outcome$ to the value $commit$, and then sends Decide messages with the positive (i.e. $commit$) indication for the transaction $XID =< req\_id, InstanceToDecide >$ to the databases. Also in this case, these messages are re-sent on a timeout basis until the acknowledgments have arrived from all the database servers.

The only case left is when the application server has found some transaction instance associated with req_id prepared at all the databases (i.e. **SCC1** holds for some transaction instance j, hence $PreparedInstance$ is defined), but it is still in doubt whether a transaction associated with the same req_id, and having instance identifier $j' < PreparedInstance$, can eventually become prepared at all the databases (i.e. **SCC2** does not currently hold). In this case, the application server sends Resolve messages (with the indication that we want to resolve doubts on instance identifiers up to $PreparedInstance$) and then re-collects again Vote messages from each DB_i with the updated $\text{MIPT}^i_{req\_id}$. (The Resolve message triggers some update operations on the corresponding MIPT at the recipient database, which mark the *null* entries with index less than $PreparedInstance$ with the $abort$ value in order to prevent the corresponding transaction instances to be eventually prepared. Hence, the Vote messages will carry MIPTs comprising the updates triggered by the Resolve message, plus any update triggered by different types of messages, e.g. Prepare messages, sent to the databases by whichever application server.) Such a resolve phase gets over when the application server detects an instance identifier for which **CC** becomes satisfied (this might happen for a $PreparedInstance$ different from the one for which **SCC1** was originally verified). At this point the final part of the ACP is executed for that instance via Decide messages, just as explained above.

As already hinted, the Request message triggering the activities at the application server might come from either the client or a database server. In case it comes from the client, the application server sends back to the client the final outcome right after the conclusion of the ACP. This also requires that

the application server assembles the result message for the client by using all the result sets associated with the committed transaction instance (i.e. $InstanceToDecide$) on all the databases (these result sets are retrieved from the $\text{MIPT}^{i}_{req\_id}[InstanceToDecide]$ received by the application server from each DB_i). On the other hand, in case the Request message comes from a database server, no reply needs to be sent back since database servers send Request messages with the only purpose to determine a final outcome for some transaction remained in the pre-commit state longer than a time-out period (see **Task 4**), and are not interested in receiving the corresponding result.

4.3 Protocol Correctness

In this section we provide a formal proof of the protocol correctness with respect to the seven Termination, Agreement and Validity e-Transaction properties, whose definition has already been recalled in Section 3.3. The correctness assumption under which the proof is carried out are presented before the proof itself. Also in this case we use auxiliary lemmas we introduce in order to simplify the structure of the proof.

4.3.1 Correctness Assumptions

Differently from the solution we presented in Chapter 3, the correctness of this protocol does not hinge on any assumption on the accuracy of the underlying failure detection scheme. This distinguishing feature of our protocol is very relevant since, as it is well-known [25], in a pure asynchronous system such as the one we consider, it is impossible to devise a failure detector's implementation providing any guarantee on its ability to correctly identify crashed processes.

The correctness of this protocol, analogously to the one in Chapter 3, depends the assumptions on that at least an application server is correct, meaning that it does not crash, and that databases are *good*, which means: (1) they always recover after crashes, and eventually stop crashing (i.e. eventually they become correct), and (2) if the application servers keep re-trying transactions, these are eventually prepared. Concerning point (1) the same considerations done in Section 3.3.1 naturally apply also in this case. Regarding point (2), it is interesting to underline that the ability to eventually commit transactions in case we keep retrying them does not contrast with the structure of our protocol, even if it allows multiple sibling transactions, associated with the same client request (i.e. tagged with the same req_id), to be concurrently active at the back-end databases (since they are activated on a timeout basis). In fact, these transactions do not block each other even in case of access to the same data, thanks to the assumed concurrency control scheme at the database

side (see Section 4.2.1). Hence, the progress of none of these transactions is indefinitely prevented due to mutual dependencies.

4.3.2 Correctness Proof

Lemma 4.3.1 *If* **CC** *is ever verified for a transaction having* $XID =< req\_id, inst\_id >$*, then* **CC** *cannot be ever verified for a transaction having* $XID' =< req\_id, inst\_id' >$*, where* $inst\_id' \neq inst\_id$*.*

Proof (By Contradiction) Assume that **CC** is ever verified for $XID =< req\_id, inst\_id >$, and that it is also found verified for $XID' =< req\_id, inst\_id' >$, where $inst\_id' \neq inst\_id$. Without loss of generality, consider the case $inst\_id' < inst\_id$. By **SCC1**, in order for **CC** to hold for $XID' =< req\_id, inst\_id' >$ it must hold that $\forall i \in [1, m]$ $\text{MIPT}^i_{req\_id}[inst\_id'].state == prepared$, whereas, by **SCC2**, in order for **CC** to hold for $XID =< req\_id, inst\_id >$ it must be $\forall j < inst\_id, \exists i \in [1, m] : \text{MIPT}^i_{req\_id}[j].state == abort$. Since $inst\_id' < inst\_id$, the latter condition implies that $\exists i \in [1, m] : \text{MIPT}^i_{req\_id}[inst\_id'].state == abort$, which would prevent **SCC1** (and hence **CC**) to hold for $< req\_id, inst\_id' >$, unless the state value of $\text{MIPT}^i_{req\_id}[inst\_id']$ had changed at some database server DB_i from *prepared* to *abort*. Since this is impossible by Observation 4.2.1, the assumption is contradicted and the claim follows. *Q.E.D.*

Lemma 4.3.2 *If an application servers sends a* Decide$[< req_id, inst_id >, commit]$ *message, the no application server ever sends a* Decide$[< req_id, inst_id >, abort]$ *message, and vice versa.*

Proof An application server sends Decide$[< req_id, inst_id >, commit]$ only if it finds that **CC** holds for the transaction $XID =< req\_id, inst\_id >$. By **SCC1**, it must hold that $\forall i \in [1, m]$ $\text{MIPT}^i_{req\_id}[inst\_id].state == prepared$. Conversely, an application server sends the message Decide$[< req_id, inst_id >, abort]$ for the same transaction only in case it verifies that $\exists i \in [1, m] : \text{MIPT}^i_{req\_id}[inst\_id].state == abort$. Since by Observation 4.2.1, no database server can ever update any MIPT entry from the *abort* to the *prepared* state value and vice versa, the previous conditions can never be both verified. Hence the claim follows.

Q.E.D.

Lemma 4.3.3 *If for a transaction* $XID =< req\_id, inst\_id >$ *a correct application server finds condition* **SCC1** *verified, this server eventually sends* Decide *messages with the commit indication for a transaction* $XID' =< req\_id, - >$ *to all the databases.*

Proof If a correct application server finds condition **SCC1** verified for some $XID =< req\_id, inst\_id >$, there are two cases:

(1) The application server also finds **SCC2** verified for $XID =< req\_id, inst\_id >$, in this case **CC** holds for $XID =< req\_id, inst\_id >$, or it finds **CC** verified for some different value $inst\_id'$. Hence, the application server sends out Decide messages with the *commit* indication to all the database servers for some $XID' =< req\_id, - >$ and the claim follows.

(2) The application server finds that **SCC2** does not hold for $XID =< req\_id, inst\_id >$ and that **CC** does not hold for any different value $inst\_id'$. In this case, the application server transmits, on a timeout basis, Resolve messages to the database servers, until Vote replies are received from all of them. Let t be the time after which the database servers are up and do not crash. After time t, since the application server is correct, the communication channels are reliable, and all the statements within the `resolve` method are non-blocking, $\text{MIPT}^i_{req\_id}$ is updated by whichever DB_i due to the Resolve message from the correct application server, and returned to this server via the Vote message. Given that after the update of $\text{MIPT}^i_{req\_id}$ on whichever DB_i, there will be no entry with index $j < inst\_id$ which still keeps the *null* value, and given that, by Observation 4.2.1, **SCC1** still holds for $inst\_id$, there are two cases:

(A) $\forall j < inst\_id, \exists i \in [1, m] : \text{MIPT}^i_{req\_id}[j].state == abort$, then **SCC2** also holds for $inst\_id$, hence **CC** is verified for $inst\_id$.

(B) $\exists j < inst\_id : \forall i \in [1, m]\ \text{MIPT}^i_{req\_id}[j].state == prepared$. Consider the minimum value of j for which previous condition is verified. We have that **SCC1** holds for j. In such a case, we also have that $\forall j' < j, \exists i \in [1, m] : \text{MIPT}^i_{req\_id}[j].state == abort$, hence **SCC2** (and thus **CC**) also holds for j.

In both cases A and B, the application server eventually sends out Decide messages with the *commit* indication to all the database servers for some $XID' =< req\_id, - >$ and the claim follows.

Q.E.D.

Lemma 4.3.4 *If a correct application server keeps on receiving* Request *messages for a given req_id, with different transaction instance identifiers, it eventually sends* Decide *messages with the commit indication for a transaction* $XID =< req\_id, - >$ *to all the database servers.*

Proof (By Contradiction) Assume that a correct application server that keeps on receiving Request messages tagged with a given req_id does not eventually send Decide messages with the *commit* indication to all the database servers for a transaction having $XID =< req\_id, - >$. Let t be the time after which the database servers stop crashing and remain up. After time t, since the correct application server keeps on receiving those requests, it will eventually send Prepare messages to all the databases, which will eventually arrive due to reliability of communication channels. Similarly, since database statements within the `vote` method are non-blocking, the databases will eventually send Vote messages, which will also eventually arrive back to the application server. After the receipt of the Vote messages there are two cases:

(A) The correct application server finds **SCC1** verified for some transaction $XID' =< req\_id, - >$. In this case, by Lemma 4.3.3 the application server eventually sends the Decide message with the *commit* indication to all the database servers tagged with $XID'' =< req\_id, - >$. Hence the assumption is contradicted and the claim follows.

(B) The correct application server finds **SCC1** verified for no value of j associated with whichever transaction identifier $XID =< req\_id, j >$. Given that the correct application server keeps on receiving Request messages tagged with req_id indefinitely, it will keep on retrying the corresponding transactions after time t. Hence, by database goodness there will be a transaction $XID =< req\_id, j >$ that can be prepared at all the databases (i.e. the `prepare` primitive does not return *abort*). Hence, the only case in which **SCC1** is not verified for $XID =< req\_id, j >$ is when for some DB_i, $\text{MIPT}_{req\_id}[j].state$ already keeps the *abort* value (so that, by the database server pseudo-code, `prepare` is not invoked by DB_i for $XID =< req\_id, j >$). However, always by the database server pseudo-code, the *abort* state value in $\text{MIPT}_{req\_id}[j].state$ can ever be set only in the following two cases:

(B.1) A Decide message with the *abort* indication from the correct application server has been received and processed by DB_i (due to the uniqueness property for XID values, no other application server can ever sent that message, since, by the application server pseudo-code, Decide messages with the *abort* indication can ever be sent only for XID associated with Request messages the server receives). However, this is impossible since, upon the verification of **SCC1** while handling whichever request with $XID =< req\_id, j >$ (i.e. right after the receipt of Vote messages from the databases), no Decide message has been sent out at all tagged with that XID value.

(B.2) A Resolve message tagged with $XID' =< req\_id, j' >$, where $j' > j$, has been ever sent from some application server, and received and processed by DB_i. In this case, that application server has found **SCC1** verified for $XID' =< req\_id, j' >$. However, given that the correct application server will keep receiving requests, retrying the corresponding transactions and recollecting Vote messages for req_id indefinitely, and given that, by Observation 4.2.1, $\text{MIPT}^i_{req\_id}[j'].state$, once set to *prepared* will not eventually change on any DB_i, the correct application server will also find **SCC1** verified for $XID' =< req\_id, j' >$. Hence we eventually fall in case (A), and the claim follows.

Q.E.D.

Termination T.1 - *If the client issues a request, then, unless it crashes, the client eventually delivers a result ([2]).*

Proof (By Contradiction) Assume by contradiction that a client issues a request, does not crash and does not eventually deliver any result. In this case, no Outcome message with the *commit* indication for a transaction associated with that request, i.e. tagged with the corresponding req_id, is received by the client. Hence, by the client pseudo-code, it keeps on sending Request messages tagged with req_id as the request identifier to the application servers indefinitely. Hence, by reliability of communication channels, a correct application server will keep on receiving Request messages tagged with that req_id by the client. By Lemma 4.3.4, this application server will eventually send Decide messages with the *commit* indication, tagged with req_id, to all the back-end databases. Also, given that there is a time t after which all the database servers stop crashing and remain up, the Decide messages are resent on a timeout basis, and the communication channels are reliable, the database servers will eventually reply with DecideACK messages tagged with req_id, which are eventually received by the correct application server. Since this server does not crash, and the Request came from a client, by the application server pseudo-code, an Outcome message, tagged with req_id, with the *commit* indication and the corresponding transaction result is eventually sent back and received by the client due to communication channel reliability. That result is therefore eventually delivered by the client since it does not

[2] As in Chapter 3, we use "vote/decide for a result" as synonymous with "vote/decide for a transaction". Similarly, "commit/abort a result" is used as synonymous with "commit/abort a transaction". Also, delivery of the result at the client side expresses that the `issue` method returns a result associated with a committed transaction.

crash. Hence the assumption is contradicted and the claim follows. $Q.E.D.$

Termination T.2 - *If any database server votes for a result, then the database server eventually commits or aborts the result.*

Proof (By Contradiction) Assume by contradiction that a database server DB_i prepares a transaction $XID =< req\_id, inst\_id >$ (i.e. votes for that result) and never decides for this transaction. This means that, even after time t, when DB_i stops crashing and remains up, it does not eventually receive any Decide message with either (A) a *commit* indication for a transaction $XID' =< req\_id, - >$ or (B) an *abort* indication for the transaction XID. In this case, by **Task 4** of the database server pseudo-code, DB_i keeps on sending Request messages tagged with req_id as request identifier to the application servers indefinitely. Hence, by reliability of communication channels, a correct application server will keep on receiving Request messages tagged with that req_id. By Lemma 4.3.4, this application server will eventually send a Decide message with the *commit* indication to DB_i for a transaction $XID'' =< req\_id, - >$. Given that we are at time after t, by channel reliability, this message is eventually received by DB_i which eventually invokes `decide` for $XID'' =< req\_id, - >$. By the specification of the `decide` primitive, DB_i decides for XID independently of the instance identifier associated with XID''. Hence, the assumption is contradicted and the claim follows. $Q.E.D.$

Agreement A.1 - *No result is delivered by the client unless the result is committed by all database servers.*

Proof The client delivers a result only when it receives an Outcome message with the *commit* indication from an application server. On the other hand, the application server sends the Outcome message with the *commit* indication to the client only after it has sent Decide messages with the *commit* decision for the corresponding transaction to all the database servers, and has received DecideACK messages from all of them. Hence all the database servers have executed the `decide` primitive with *commit* as input parameter. Therefore, all we need to prove is that, prior to the execution of `decide` with the *commit* indication (A) that transaction had been prepared and (B) had not been aborted. Point (A) straightforwardly follows from **SCC1**, and by that a MIPT entry can store a *prepared* state value for that transaction only if the `prepare` primitive succeeds in preparing the transaction. Concern-

ing point (B), a transaction $XID =< req\_id, inst\_id >$ can be aborted by a database server either because the database server receives a Decide message with *commit* indication for a transaction having $XID' =< req\_id, inst\_id' >$, where $inst\_id \neq inst\_id'$, which is excluded by Lemma 4.3.1, or because the database server receives a Decide message with the *abort* indication for that same transaction, which is excluded by Lemma 4.3.2. Hence the claim follows. *Q.E.D.*

Agreement A.2 - *No database server commits two different results*

Proof A database server can commit two different results, i.e. two different transactions $XID =< req\_id, inst\_id >$ and $XID' =< req\_id, inst\_id' >$ associated with the same client request only, if it receives a Decide$[< req_id, inst_id >, commit]$ and a Decide$[< req_id, inst_id' >, commit]$ where $inst\_id \neq inst\_id'$. In this case, some application server must have sent the Decide messages with the *commit* indication for the two different transactions $XID =< req\_id, inst\_id >$ and $XID' =< req\_id, inst\_id' >$. Hence, the application servers must have found **CC** verified for both $XID =< req\_id, inst\_id >$ and $XID' =< req\_id, inst\_id' >$. However, this is impossible by Lemma 4.3.1. *Q.E.D.*

Agreement A.3 - *No two database servers decide differently on the same result.*

Proof A database can decide *commit* for a result only if it receives a Decide message with the *commit* indication for the corresponding transaction, whereas it can decide *abort* only if it receives a Decide message with the *abort* indication for that same transaction or a Decide message with a *commit* indication for a different transaction associated with the same *req_id*. By Lemma 4.3.2, it follows that no two database servers can ever receive Decide messages with a *commit* indication for two different transactions associated with the same *req_id*. Finally, by Lemma 4.3.3 no two application servers can send a Decide message with a *commit* indication and a Decide message with an *abort* indication for the same transaction XID. Hence the claim follows. *Q.E.D.*

Validity V.1 - *If the client delivers a result, then the result must have been computed by an application server with, as a parameter, a request issued by the client.*

Proof The client delivers the result after it receives the Outcome message with the *commit* indication. Such a message is sent to the client by some application server if, and only if, the application server has received from each database server an Outcome message with the *commit* indication for that result. This happens only in case the result has already been committed.

Given that a transaction associated with an identifier is committed only after an application server has computed and prepared it on all the databases, there are two cases: (i) The application server computes and prepares the transaction after it has received the Request message with that identifier from the client, in this case the claim trivially follows; (ii) The application server computes and prepares the transaction after it has received the Request message with that identifier from a back-end database server. In this case, given that the database server does not spontaneously issue requests, it means that an application server must have previously received a Request message with that same request identifier from the client. Also in this case, therefore, the claim follows. *Q.E.D.*

Validity V.2 - *No database server commits a result unless all database servers have voted yes for that result.*

Proof A database server commits a result only after it receives the Decide message for the corresponding transaction, with a *commit* indication. On the other hand, by **SCC1**, an application server can send such a message only after every having verified that all the database servers have a MIPT entry for that result storing the *prepared* state value. By the database server pseudo-code, a MIPT entry with *prepared* state value is stored only after the database server prepared (i.e. voted positively) for that transaction. Thus, if a database server commits a transaction, then all database servers must have prepared that same transaction. *Q.E.D.*

4.4 On Practical Issues

In this section, analogously to what we did in Section 3.4, we discuss the practical issues of how to avoid unbounded growth of recovery information logged by the database servers, and of how to cope with application logics that execute transactions spanning only a subset of the back-end servers.

For what concerns garbage collection of unneeded recovery information, this protocol poses an additional difficulty with respect to the protocol described in Chapter 3. This is essentially due to that, while in our former proposal the client is the only process to generate and retransmit new request

instances, in this latter protocol the database servers are as well allowed to send out new request instances if a transaction were to remain locally prepared for an excessively long period of time. Consequently, we are here faced with the additional problem of ensuring that, following the removal of the MIPTs at the databases server, no Request message (possibly) originated by a database gets asynchronously processed giving raise to a (new) transaction that gets eventually committed.

However, in the most likely scenario in which neither the client nor the databases experience timeout expirations (and therefore do not submit additional requests instances for a same client request), one could rely on the following simple mechanism to allow safe removal of logged MIPTs. Database servers should piggyback on the DecideACK message with the *commit* indication directed to the application server an additional flag conveying information on that they did not perform any request retransmission associated with that same request identifier. The client should also send an explicit acknowledgment message to the application server upon receipt of the Outcome message. At this point, the application server would simply notify the databases to discard the corresponding MIPTs. This is safe given that neither the client nor the database servers will ever re-transmit that request and that the only transmitted Request message has been already processed by some application server.

In case the client or the database server had to give rise to any request retransmissions, an approach analogous to the one described in Section 3.4, could be adopted to reduce the amount of space occupied by the stored MIPTs at the database servers. Essentially, the above described acknowledgment scheme could be modified as follows: the client, as well as the database servers, could send to the application server a different acknowledgment message to signal that they have performed some request retransmission. In this case the application server could inform the database servers to discard the transaction result (while maintaining the information on the transaction state) from every entry of the MIPT associated with that request. As already discussed in Section 3.4, this would allow to release most of the storage resources occupied by the recovery information. Finally, just like for the protocol presented in Chapter 3, in practical settings one could still rely on appropriately selected TTL values, after which deletion of the MIPTs from the databases could be performed.

For what concerns the issue of how to deal with client requests whose processing requires transactional access only to a subset of the back-end databases, there are two cases to consider.

If the identities of the involved databases were determined as a deterministic function of the client request content, no additional mechanisms be necessary given that every application server processing an instance of the same

client request would naturally interact with the same subset of databases.

Conversely, if the business logic residing at the application server selected the databases to be accessed in a non-deterministic manner, additional mechanisms, similar in spirit to the one described in Section 3.4, would be required in order to ensure that application servers processing different instances of the same client request were forced to evaluate conditions **SCC1** and **SCC2** over the same set of databases before attempting to commit a transaction.

Finally, if stable storage capabilities were allowed at the client side, our protocol could be extended to permit correct recovery of clients after crashes in an analogous manner to the one already discussed in Section 3.4. The client should simply log the Request message sent out for each issued request, as well as the first received Outcome message carrying a *commit* indication. Upon recovery, if no Outcome message is logged with the *commit* indication, the client should re-transmit a Request message tagged with the same request identifier and an increased *instance_number* value.

4.5 Overhead and Integration with Conventional Technology

In this section we comparatively evaluate the overhead of our protocol vs other solutions. Then, we discuss issues concerning its integration with Commercial Off-The-Shelf (COTS) systems. We compare the protocol with the same state of the art solutions considered in Chapter 3, namely:

- The baseline protocol;
- The persistent queue approach (PQ) [16]; and
- The coordination-based schemes AR and PBR presented, respectively, in [44] and [45].

The behavior of these protocols was already detailedly described in Section 3.5 and schematized in Figure 3.6. Additionally, we consider in the comparison also our own solution presented in Chapter 3. To distinguish between that solution and the protocol presented in this chapter we use, respectively, the notations Proto-A and Proto-B. Their comparative schematization is shown in Figure 4.5.

Given that these two protocols exhibit the same message exchange pattern and, that both of them avoid additional interactions with remote components (namely the queuing system for PQ and the other application server replicas for AR and PBR), the analytical model of Proto-A presented in Section 3.5 would essentially hold also for Proto-B, except for minor modifications of the

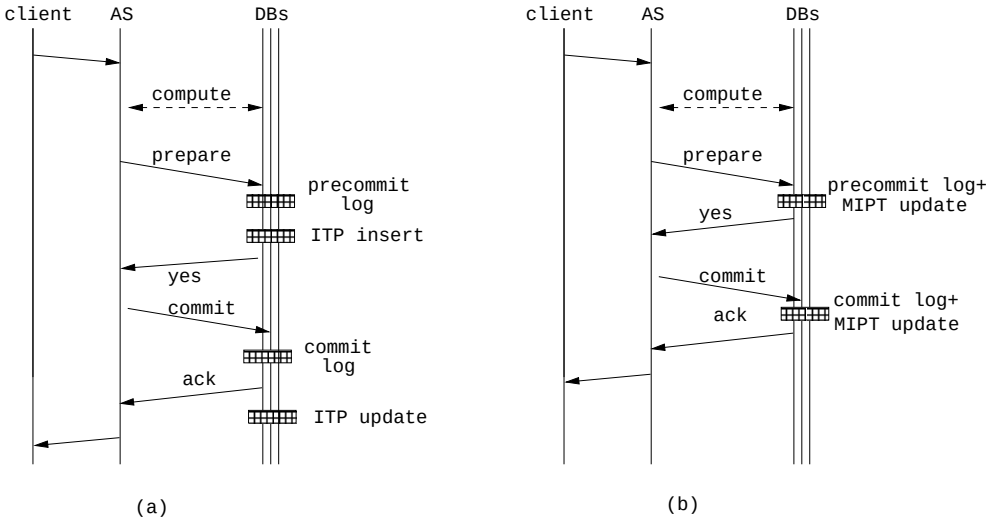


Figure 4.5: Schematization of the (Normal) Behavior of Proto-A (a) and Proto-B (b). (Filled boxes represent eager disk accesses)

values of some specific terms expressing local processing actions at the involved processes. Hence, response time results analogous to the one achieved for Proto-A in Section 3.5 would be obtained for Proto-B when using that same analytical approach and parameterization in the comparison. (This also means that Proto-B exhibits the same advantages of Proto-A compared to the other solutions, in terms of performance effectiveness especially in large scale systems.) For this reason, in this section we choose to compare the considered protocols at a higher level of abstraction. This is done by relying on two classical metrics for distributed transaction processing, namely the number of (server-side) message rounds needed before returning a response to the client, and the number of required eager logs. Table 4.1 reports the corresponding values for each of the considered protocols in case of a (likely) nice-run, not incurring failures or suspect of failures. These values clearly show how Proto-B achieves end-to-end reliability guarantees at the same cost of a baseline protocol tolerating only database server failures.

It is also interesting to highlight that, when directly comparing our own solutions, Proto-B achieves further overhead reduction compared to Proto-A. This is achieved by saving the cost of an eager log at the back-end databases. Specifically, as shown in Figure 4.5, Proto-A performs the insertion of the ITP within an external transaction that is independent of the distributed transaction coordinated by the application server, and whose commitment requires an additional eager access to persistent storage. It is worthy underlining that,

	# message rounds	# eager logs
baseline	2	2
PQ	4	5
PBR & AR	4	2
Proto-A	2	3
Proto-B	2	2

Table 4.1: Protocols Comparison.

rather than for theoretical reasons, performing the ITP manipulation out of the scope of the application level distributed transaction is related on the pragmatical design reason of ensuring Proto-A fully compliance with standard distributed transaction processing technology. Manipulating the ITP directly within the application level distributed transaction (e.g. by including the SQL statements operating on the ITP table within the same transactional context of the business logic transaction) would in fact require a modification of the conventional database logic providing support for 2PC. This is due to that, upon entering the pre-commit state for a transaction, a DBMS maintains durable locks on any data item updated by that transaction, and consequently also on the corresponding ITP tuple, until it does not receive a *commit/abort* decision for the transaction. In the meanwhile, it would be therefore impossible for any application server to test the transaction outcome through a Resolve message, as both the `insert` and `lookup` primitives in the `ITPLogger`'s `try_abort` method would be forced to block. With such a blocking scenario arbitrarily protracting in case of crash (with no recovery) of the application server that had originally prepared the blocking transaction.

On the other hand, Protocol-B has two essential building blocks in the innovative, ad-hoc MIP concurrency control and transaction demarcation schemes, whose integration within existing DBMS products mandatorily requires alteration of their inner logic. On the other hand, this gives us more space for optimizing the database treatment of recovery information. Specifically, we may rely on custom mechanisms performing the MIPT's manipulations within the same context of the application level distributed transaction, while releasing locks on the MIPT as soon as its manipulation (occurring within the `vote` and `resolve` methods) ends, thus avoiding blocking situations.

Along the lines of the previous observations, we conclude this chapter by discussing some practical aspects an implementor would face when adopting our solution. For what concerns the implementation of the client retransmission logic, Proto-B does not pose any practical difficulty ([3]). The former could be straightforwardly supported by a Web browser by relying on, e.g., a Java

[3] Actually these same considerations also apply to Proto-A.

applet, Javascript technology or by exploiting browser proprietary technology, such as ActiveX in Microsoft's IE or ad-hoc developed extensions in Mozilla's Firefox.

Regarding the implementation of the middle-tier server logic, one could exploit the strong trend exhibited by modern application server to be implemented on top of off-the-shelf industry middleware frameworks (e.g. Sun Microsystems' Java 2 Platform Enterprise Edition - J2EE, and Microsoft's .NET). These permit assembly of services from reusable components, relying upon container environments to provide commonly required support for naming, communication, security, clustering, persistence, and transactions. In addition to providing an integrated environment for component execution, which significantly reduces the time to design, implement, and deploy applications, such frameworks incorporate "best practices" designs. The latter provide developers with design patterns, suggesting a standardized structure upon which distributed component-based systems should be based [72]. For what concerns transaction management, current industry standard middlewares already embed ad-hoc services, such as the well-known JTS for the J2EE platform, providing the abstraction of "Container Managed Transactions" [114]. This approach aims at saving application developers from the burden of implementing low level, critical and error-prone mechanisms, such as demarcation and coordination of distributed transactions. In such a context, delegating the middleware container to host our protocol's application server logic (which can be basically viewed as a non-conventional, MIPT-based transaction coordination scheme) appears as the most natural choice.

Finally, for what concerns the integration of the MIP scheme with COTS database systems, in order to support, e.g., the ad-hoc transaction demarcation and concurrency control mechanisms (see Section 4.2.1), as said we need to alter the inner logic of the underlying DBMS. The hurdles an implementor should face while developing a MIP implementation are strictly related to the design choices of the specific DBMS. Anyway, MIP implementations could be relatively easily developed in case of integration with DBMS relying on data-item versioning for concurrency control purposes. Specifically, multiversion databases (see, e.g., [118]), have the ability to maintain multiple versions of a same data-item, so that concurrency control selects which version must be supplied for a given read operation by a certain transaction [14]. Although this approach is orthogonal to our proposal (since multiversion concurrency control aims at increasing the concurrency level among independent transactions by letting them access different versions of the same data items), it could be anyway used as the basis for the concurrency control scheme required to support the MIP semantic (which aims at increasing the concurrency level only among sibling transactions associated with the same client request).

Chapter 5

Coping with Performance Failures

Beyond traditional crash failures, we can envisage a wider type of failure class, we can term as "performance failure", which is very closely related to Quality-of-Service (QoS) oriented, advanced system design. We say that a performance failure occurs in a multi-tier application when the completion of request processing and result delivery at the client side is delayed due to:

(i) A crash of some component; and/or

(ii) Poor responsiveness of some component due to, e.g., overload/congestion situations.

We note that performance failures (independently of their origin) constitute a relevant problem in real life applications. The user's perceived response time is in fact one of the main issue for differentiation among electronic services, such as, e.g., e-Commerce or on-line stock trading applications, since it directly determines the level of user's satisfaction [19]. In the context of e-Commerce applications, for example, a striking result shown in [130] is that an increase of the user perceived response times can rapidly lead to the devastating phenomenon of excessive users abandon rate. The empirical study conducted in this work highlighted that while the abandon rate remains modest (i.e. under the 2%) if the response time is under the threshold value of 7 seconds, it dramatically increases, up to 70% in case of a few additional seconds of delay in the delivery of the output at the client side. In a broader sense, response time is a critical factor that can have detrimental effects on the business process supported by the growing number of real-time applications that are nowadays accessible over the Internet, such as, e.g., on-line auctions or stock/equity trading systems.

We argue that an effective approach to tackle performance failures would be to mask the original cause of the failure (either crash or overload/congestion), by activating the processing of client requests in parallel along different chains of components (as much uncorrelated as possible), so to let end users perceive the latency of the quickest interaction and increase the likelihood to timely deliver responses. Interestingly, the protocol described in Chapter 4 does allow multiple instances of the same client request to be (possibly) processed in parallel by different application server replicas, reconciling them at commit time. (In fact, in that protocol fail-over is performed on the basis of simple timeout based request retransmission logic, and not via extermination of previously submitted requests. Also, it avoids mutual blocking situations among the different request instances, due to data conflicts and pre-commit locks, thanks to the innovative MIP scheme for distributed transaction management.) Hence, it represents a natural building block to support a parallel invocation scheme aimed at tackling not only classical crashes failures, but also performance failures.

This Chapter addresses exactly the latter issue, showing how the protocol in Chapter 4 can be naturally transmuted into a parallel invocation scheme. We motivate, propose and evaluate the adoption of such a parallel invocation scheme in the context of large scale service delivery platforms, where the correlation of different chains of components toward the back-end servers is typically strongly reduced thanks to the large redundancy of geographically distributed middle-tier server replicas and to the path diversity provided by the underlying (multi-hop) network infrastructure [8, 28].

As we will discuss, this is, at the best of our knowledge, the first parallel invocation scheme ever proposed in literature to address the relevant scenario of business logics performing non-idempotent manipulations (e.g. updates) of transactional data in a multi-tier system with stateless application servers.

5.1 Deriving a Parallel Invocation Protocol

This section is devoted to formalize the changes required to transmute the protocol in Chapter 4 into a parallel invocation protocol suited for coping with performance failures.

While presenting the parallel invocation protocol, we only focus on the presentation of the client behavior and of the database server behavior since the application server behavior remains identical among the two protocols. For simplicity of presentation, we assume that the number of application server replicas contacted in parallel by the clients is an a-priori known constant value, which we refer to as $ParallelismLevel$. However, we will later show how to avoid such an assumption via marginal modifications of the payload of the

```
Class Client{
  List ASlist = {AS_1, ..., AS_n}; ApplicationServer AS; Result result;
  Outcome outcome; RequestIdentifier req_id; InstanceIdentifier inst_id;
  Counter counter = InitialValue;

  Result issue(Request req){
   outcome = abort;
   req_id = SetId(req);
   while(true){
    set TIMEOUT;
    for (int i = 0; i < ParallelismLevel; i++) {
      AS = ASlist.next();
      inst_id =< GetMyCategory(), ++counter >;
      send Request[req, < req_id, inst_id >] to AS;
    }
    wait (receive Outcome[< req_id, − >, commit, result] from
          any application server in ASlist) or (TIMEOUT);
    if (received Outcome[< req_id, − >, commit, result]) return result;
   } // end while
  } // end issue
} // end class
```

Figure 5.1: Client Behavior within the Parallel Invocation Protocol.

exchanged messages.

5.1.1 Client Behavior

Figure 5.1 shows a revised version of the client pseudo-code originally presented in Figure 4.3 of Chapter 4, updated to reflect the presence of parallel invocation.

The client multicasts its request to a number of *ParallelismLevel* different application servers chosen from *ASlist*, and waits for the first Outcome message reporting a *commit* indication from any of them, or for a timeout expiration. In the former case, it returns the corresponding *result*. Otherwise, the client keeps on multicasting its request to a different set of *ParallelismLevel* application servers. The employment of the retransmission logic is obviously meant to tackle with the scenarios in which every contacted application server either reports an *abort* outcome (e.g. because the database servers refused to commit their transactions) or simply does not timely deliver a reply (e.g. because all of them prematurely crashed). In other words, the latter ones are the situations in which even the employment of the parallel invocation was unable to effectively tackle performance failures in order to guarantee successful, timely completion of the whole end-to-end interaction. Note that, just like in the protocol in Chapter 4, each instance of a request message transmitted by the client is tagged

with the same *req_id* (and *category*), but with different, progressive values for *instance_number*. It trivially follows that the first $ParallelismLevel$ request instances initially multicast by the client will have instance numbers in the range $[InitialValue, InitialValue + ParallelismLevel)$.

5.1.2 Database Server Behavior

In Figure 5.2 we specify the pseudo-code for the behavior of database servers. Actually, we do not report the whole pseudo-code, but only the modifications applied to the original database server pseudo-code in Chapter 4. These are highlighted via a box, and are restricted to the Prepare method. Such modifications are meant to maximize the performance benefits achievable through the parallel invocation scheme, and specifically, to avoid additional communication rounds (with respect to the standard two phases of 2PC) in between application and database servers for the processing of transactions having $inst\_id =< ClientCategory, j >$ with instance number $j \in [InitialValue, InitialValue + ParallelismLevel)$ (i.e. associated with the first set of $ParallelismLevel$ requests multicast by the client). In fact, as discussed in Section 4.2.3, the original protocol in Chapter 4 necessitates two communication rounds for the processing of the first request spawned by a client, i.e. the one associated with $instance\_number = InitialValue$, but may require an additional communication round for the exchange of Resolve/Vote messages (between application and database servers) when dealing with transactions associated with subsequent client request retransmissions, i.e. having $instance\_number > InitialValue$. However, this feature is undesirable for the parallel invocation protocol, as it would a-priori penalize (in terms of performance) the processing of requests having $instance\_number \in [InitialValue + 1, InitialValue + ParallelismLevel)$ with respect to the request having $instance\_number = InitialValue$.

In order to cope with the latter performance issue, the database server treats in an ad-hoc manner the transactions having instance identifier $inst\_id' =< ClientCategory, j >$ with instance number $j \in [InitialValue + 1, InitialValue + ParallelismLevel)$, i.e. all the transactions associated with the first set of requests multicast by the client except the one having $instance\_number = InitialValue$. When asked to prepare one of these transactions, the database server executes the `resolve` method with $inst\_id'$ as input parameter immediately after having successfully invoked the `prepare` primitive, rather than waiting for an explicit Resolve message from the application server as in the original protocol version. This ensures that any sibling transaction having $inst\_id < inst\_id'$ and not yet prepared, will never be able to prepare in the future at that database (being its MIPT entry marked with the *abort* state value). As a direct consequence, if we denote

```
TypeMIPT DatabaseServer::vote(Request req, XID < req_id, inst_id >, Result result){
  atomically do{
    if (MIPT_req_id does not exist) {create MIPT_req_id; MIPT_req_id.req = req;}
    if (MIPT_req_id[inst_id].state == null) {
      state = prepare(req_id, inst_id);
      if (state == prepared) {
        MIPT_req_id[inst_id].(state, result) = (prepared, result);
        if ( inst_id =< ClientCategory, j > and
             j ∈ [InitialValue + 1, InitialValue + ParallelismLevel) )
               resolve(req_id, inst_id);
      } //end if (state == prepared)
      else MIPT_req_id[inst_id].(state, result) = (abort, null);
    }// end if
  } // end of atomic statement
  return MIPT_req_id;
} // end vote
```

Figure 5.2: Database Server Behavior within the Parallel Invocation Protocol.

with $minPrecommitted$ the minimum transaction instance identifier that an application server, processing a transaction associated with the first set of requests multicast by the client, were to find pre-committed at all the databases after collecting the Vote messages (i.e. the minimum instance identifier for which **SCC1** were found to hold), then it is guaranteed that every transaction having $inst\_id < minPrecommitted$ will be marked as aborted in at least one database MIPT. Hence both **SCC1** and **SCC2** will simultaneously hold for $inst\_id = minPrecommitted$ and the application server will not have to activate any additional communication round.

To help clarifying discussion, in Figure 5.3 and Figure 5.4 we contrast the processing of a transaction associated with a client transmitted request having $instance\_number = InitialValue + 2$ according to, respectively, 1) the original database server pseudo-code presented in Section 4.2.1 and 2) the above described database server pseudo-code assuming $ParallelismLevel > 2$.

It is important to underline that if we did not restrict such an ad-hoc treatment to the transactions having instance identifier minor than $< ClientCategory, InitialValue + ParallelismLevel >$, we could in theory give rise to the scenario in which, due to an unlucky interleaving of Prepare messages at the database servers, transactions end up indefinitely aborting each other, hence preventing any transaction from getting ever pre-committed at all the databases (see Figure 5.5). In its turn, this would indefinitely avert committing transactions associated with the client request and consequently the delivery of a result to the client, thus leading to a violation of the **T.1** e-Transaction property. However, by bounding to $ParallelismLevel$ the num-

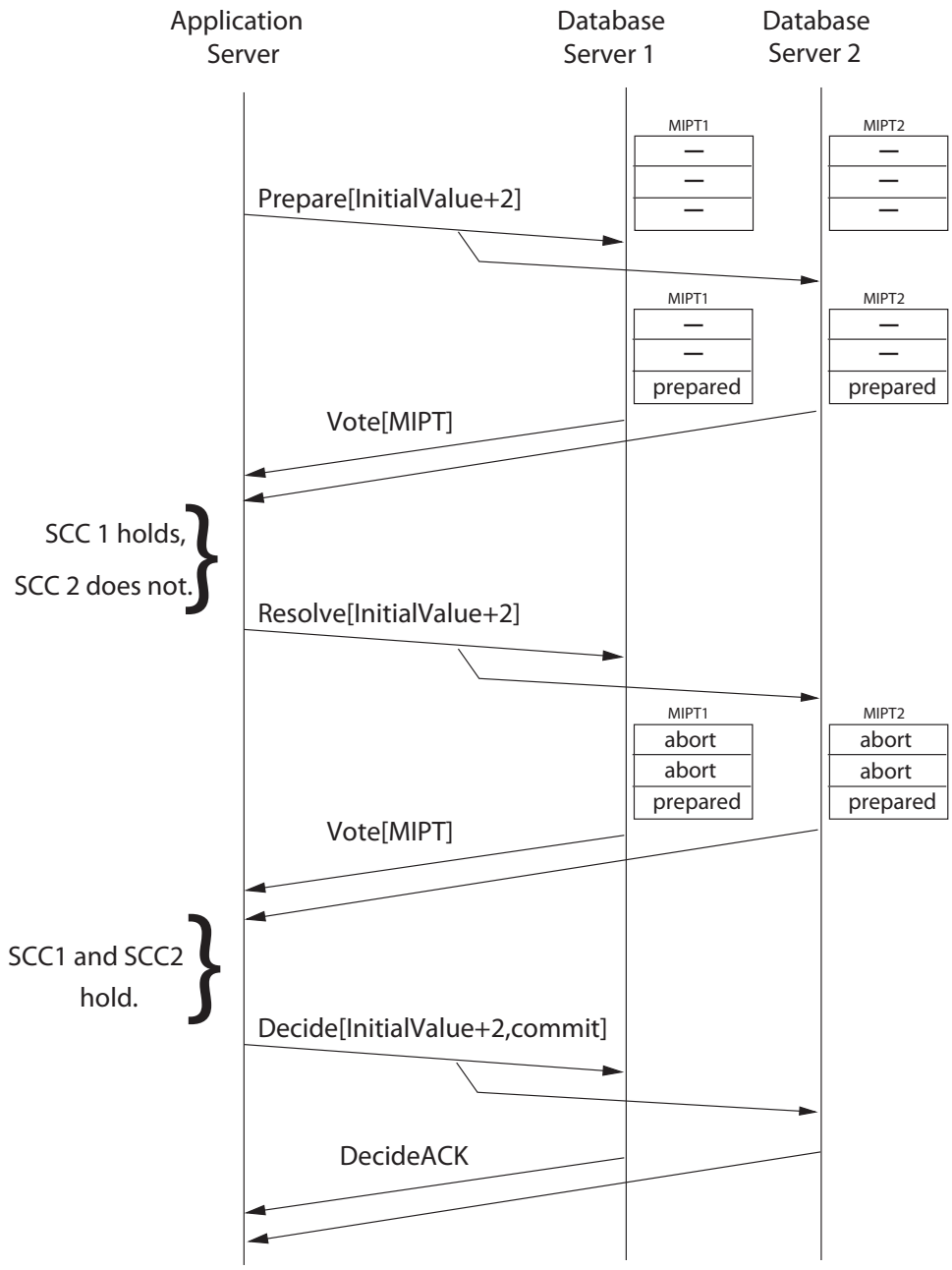


Figure 5.3: Example Execution of a Transaction Associated with a (Client Transmitted) Request Having $instance\_number = InitialValue+2$ According to the Database Server Pseudo-code in Figure 4.2. (Only relevant identifiers are shown.)

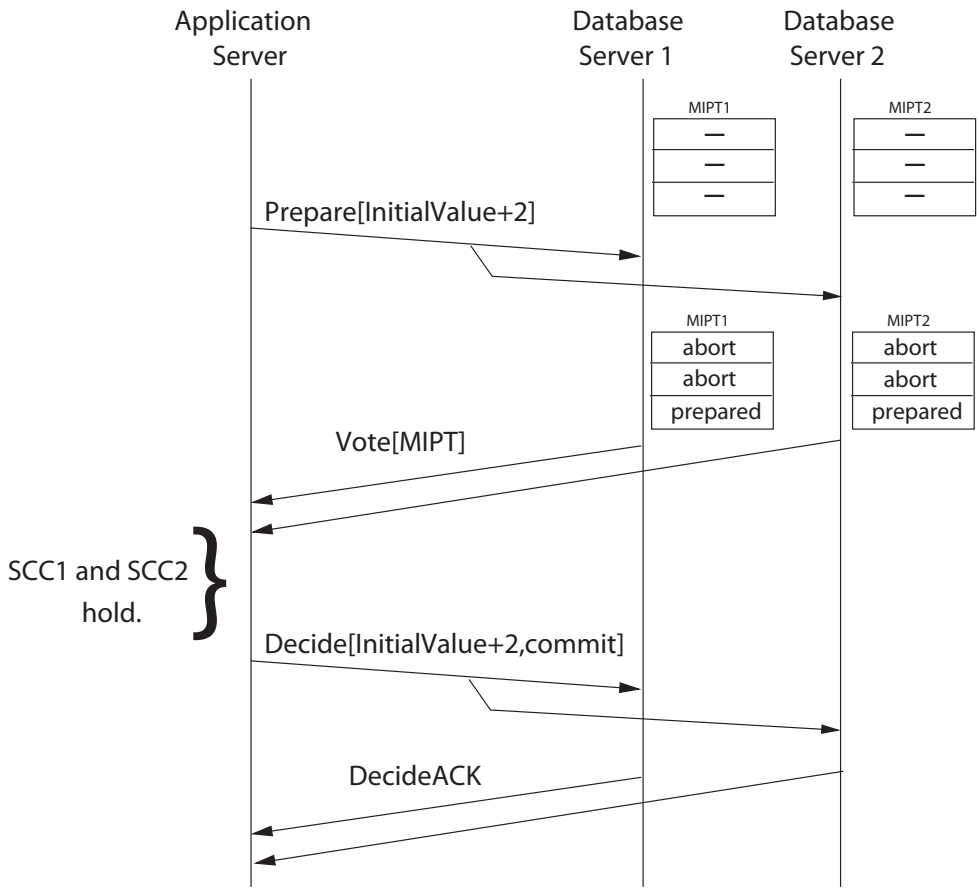


Figure 5.4: Example Execution of a Transaction Associated with a (Client Transmitted) Request Having $instance\_number = InitialValue + 2$ According to the Database Server Pseudo-code in Figure 5.2 and Assuming $ParallelismLevel > 2$. (Only relevant identifiers are shown.)

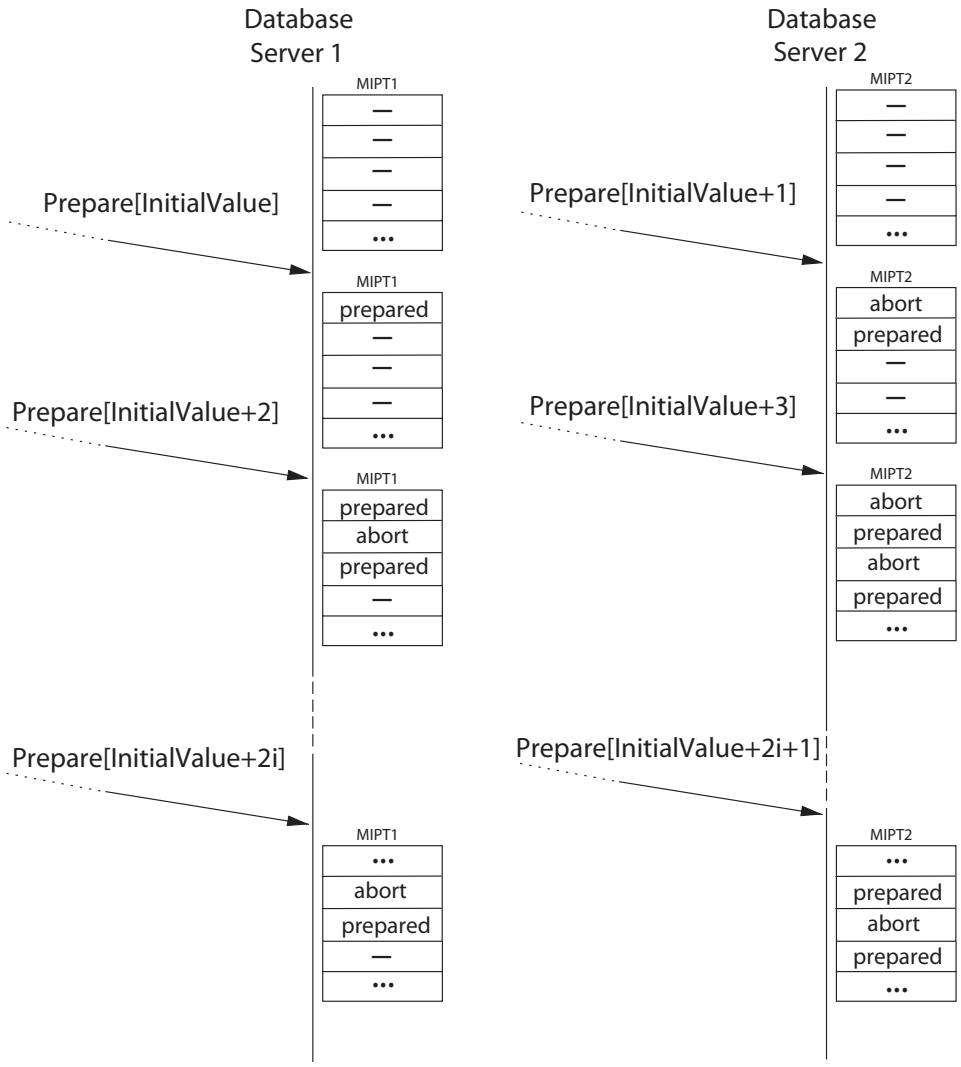


Figure 5.5: Interleaving of Prepare Messages at the Database Servers Giving Rise to an Unbounded Sequence of Mutual Aborts among Sibling Transactions Assuming $ParallelismLevel = \infty$. (Only Prepare messages and relevant identifiers are shown.)

ber of transactions for which we autonomously execute the `Resolve` method at the database side, we fall back to the original protocol treatment as soon as the client or the back-end databases (experience a timeout and) retransmit the request. This prevents *unbounded* mutual abort of sibling transactions and allows us to preserve the whole set of e-Transaction guarantees.

As a final consideration, in order to remove the assumption that the *ParallelismLevel* value is a-priori known by both the clients and the database servers, all we would need to do is to have the client dynamically determine such a value (e.g. by taking it as an additional input parameter of the *issue* method) and piggyback it as a field of the request content. This would permit to the database servers to differentiate the treatment of transactions, as described above, on the basis of this information.

5.2 Innovation vs State of the Art Approaches

Over the years, a number of solutions, e.g., [108, 97, 80, 27, 22, 54], have been proposed relying on the idea of simultaneously exploiting multiple network paths among communicating parties in order to provide enhanced performance and reliability. The common base underlying these approaches is to leverage the inherent path-diversity of multi-hop networks so to reduce the likelihood to incur link congestions or failures. Dispersity Routing [80] and IDA [97] were probably some of the first works in the area of path-diversity. They essentially proposed to split the transfer of information over multiple network paths so to provide enhanced dependability and performance. Simulation results and analytic studies [9, 18] have later shown the benefits of this approach in the context of real-time communications. More recently, the notion of path-diversity has been exploited to achieve higher QoS levels in content delivery applications, such as parallel file downloads [30, 22, 28], cooperative Web cache sharing [126, 61], multimedia streaming [8] and access to non-transactional Web-services [83]. Generally speaking, a promising result highlighted in some of these works [8, 28] is that existing content/service delivery infrastructures seem to have the intrinsic potential for providing uncorrelated network paths among a client and multiple edge servers, even though these infrastructures were originally designed to minimize distance from clients to edge servers rather than for maximizing path disjointness. Compared to all these approaches, our parallel invocation protocol has the distinguishing feature of coping with requests that perform transactional manipulations (e.g. updates) of application data which requires apposite mechanisms to address the additional difficulties arising with respect to content delivery applications (e.g., non-idempotence of requests processing or mutual blocking of sibling transactions due to data conflicts).

As a matter of fact, our parallel invocation protocol represents a solution orthogonal to all the approaches relying on pervasive caching of both static contents, e.g., [121, 36], and DBMS data, e.g., [84, 33, 69], at the edge server side, which are aimed at tackling performance failures in large scale service delivery infrastructures. These approaches effectively cope with performance failures (due to server overloads and/or network congestions) for the case of client requests performing read only access to (dynamic) application data. Instead, as pointed out above, our parallel invocation scheme copes with transactional requests, that, for a number of relevant reasons, need direct access to primary data copies at the back-end tier. These reasons can be schematized as follows:

- Often it is the very same nature of the application to impose interactions with third-party services, as in the case, e.g., of e-Shops relying on trusted authorities for validation of electronic payments.
- Further, data replication over third-party owned, geographically dispersed edge servers may not be a viable option in very common applications due to privacy and security considerations, as in the case, e.g., of e-Health applications manipulating sensitive information on personal health conditions and practices.
- However, even in those contexts where data replication represents a feasible opportunity, the de facto standard approach [51] to transactional WAN data replication is to rely on some primary copy scheme, which impose redirecting update requests to (remote) primary databases. This depends on that update anywhere-anytime-anyway transactional replication strategies, such as, e.g., [3, 93, 107, 122], are known to provide good performance when the degree of replication is kept on the order of a dozen of sites, but have unstable behavior as the number of replicas and workload scale up [51], hence revealing ineffective in large scale infrastructures composed by tens of thousands of servers.

Overall, our parallel invocation protocol targets precisely these scenarios (i.e. applications performing update manipulations of remote transactional resources), thus complementing existing data replication techniques by masking performance failures in contexts where these latter solutions would reveal either not viable or ineffective.

Parallel invocation schemes are also at the basis of long established active replication techniques [104, 103]. However, our approach assumes a different architectural model and tackles an orthogonal problem with diverse techniques. Specifically, classical active replication exploits atomic multicast protocols [35, 99] to ensure consistent evolution of the state trajectory of a set

of stateful, deterministic server replicas. Instead, in our parallel invocation protocol, clients multicast (in a non-atomic fashion) their requests to a set of stateless, non-deterministic application server replicas so to pursue timely delivery of results in presence of middle-tier servers' crashes/overloads and/or network congestions. Also, our protocol avoids the inherent costs of atomic broadcast protocols, which have been shown to suffer from significant performance degradations in WANs [6] due to relatively high loss rate and variations in message propagation time.

Finally, we remark that many results exist for what concerns the design and implementation of data centers (e.g. database systems) offering facilities for supporting high performance, or even real-time applications. These facilities span from (i) solutions for ad-hoc management of transactions with time constraints (e.g. [112, 65, 64, 92, 109, 124]), to (ii) solutions aiming at reducing the impact of the latency of distributed commit schemes on the availability of pre-committed data (e.g. [34, 58]), to (iii) solutions for availability and reliability of data centers (e.g. via replication and ad-hoc recovery schemes for critical data, see [106, 129, 123]). Even though some of these solutions (e.g. [109]) have been proposed in the context of Web applications, they do not straightforwardly fit all the timing requirements of a whole end-to-end interaction along all the components of modern multi-tier applications, which is the issue we aim at addressing, in an orthogonal way, via our parallel invocation scheme.

5.3 Simulation Study

In this section we report the results of an extended simulation study devoted at evaluating the impact of our parallel invocation protocol from a twofold perspective:

1. Quantifying the benefits that can be achieved in terms of reduction of user perceived latencies compared to a classical protocol for Web based transactional applications not exploiting parallelism.

2. Evaluate possible variations in the system throughput due to the concurrent activations of multiple transactions at the back-end databases on behalf of the same client request.

In other words, we study the impact of our protocol on performance from both the end-user's and the server side's perspective. With no loss of generality, the simulation study will be focused on the case of performance failures caused by overload/congestion of the network, which is the commonly recognized more likely scenario for this type of failures (hence representing an adequate test bed). Also, we focus on the case of ADN like infrastructures,

for which, as discussed in the previous section, our proposal represents a complementary approach to well established techniques addressing performance issues.

5.3.1 Network Model

As highlighted in a number of previous studies [4, 8], the effectiveness of any parallel invocation scheme strongly depends on the actual disjointness among the simultaneously explored paths, which determines the level of disjointness of the chains of components involved in the processing of the client request.

To determine how our proposal fares in different networks, we took an approach similar to the one used in [8]. In our experiments, we examined both the BRITE [82] generated topology (obtained by using the standard settings of the BRITE topology generator) and the NLANR [89] graph, representative of connectivity among real Internet autonomous systems at the latest available date, namely January 2000.

To assign the client, edge server and data center roles to a subset of the nodes in the topologies, we used a placement algorithm based on the connectivity degree of nodes:

- *Edge Servers:* To emulate edge server location proper of ADN infrastructures, we placed middle-tier application servers at the edges of a topology, where edges are defined as nodes with degree of two or three.

- *Data Centers:* To emulate the location of data centers, hosting back-end databases, at the most connected part of a network, we placed data centers at the core nodes of the topology, which we define as nodes with the highest degrees.

- *Clients:* To emulate client location at the furthest edge of a topology, clients were randomly chosen among those nodes having degree of one.

Obviously the ideal case would be to use a real server location graph from an ADN company, but such information is proprietary and not available, which is the reason why we chose to rely on this simple placement algorithm inspired by the one presented in [8] in the context of Content Delivery Networks (CDNs) based video-streaming.

To generate realistic values for the network latencies perceived by the hosts participating in our protocol, under both normal and anomalous (e.g. congested) situations, the considered topologies were complemented by both mathematical models and publicly available empirical measurements of Internet latencies.

For what concerns the packet loss model across the links, we chose the widely adopted two-state Gilbert model parameterized by transition probabilities $\{p,q\}$ where p is the probability of going from no loss state to loss state, and q is the probability of going from loss to no loss. The Gilbert model is widely used to model bursty traffic for its simplicity and mathematical tractability. Like in several other studies, e.g. [8], we assumed for simplicity that faults over each link can be modeled as independent.

In order to accurately determine the message transfer time over TCP connections in presence of packet losses, we adopted the TCP analytical model in [24]. This model provides accurate estimations of TCP transfer times on the basis of (i) the number of TCP fragments to be sent (i.e. the message size), (ii) the expected number of packet losses, and (iii) the end-to-end RTT latency. Given that a number of studies (e.g. [10]) have shown that WWW traffic exhibits heavy-tailed message size distributions, our simulator determines the message size according to a Pareto distribution. The end-to-end RTT for each message transmission is derived by means of the RTT probability distribution shown in [1], that was empirically obtained at the light of the RTT measurements carried out between the NASA's Glenn Research Center Web Server and its clients. These RTTs are representative of end-to-end network latency between hosts communicating across the Internet. In order to correlate the length (in terms of number of hops) of a path in a topology with the corresponding end-to-end RTT value, we determined the RTT on each link over which packets are transmitted by scaling (dividing) the end-to-end value by the average path length.

Note that in practice a strong correlation exists between a link RTT and the occurrence of packet losses over that link. In fact, the RTT values are comprehensive of router queuing delays, which are likely to be large in case of packet losses (since losses are typically due to the excessive growth of routers queues). In order to capture such a correlation in our simulator, in absence of packet losses we randomly pick the current link RTT from the first half of the empirical RTT distribution, namely the half collecting the lowest measured RTT values. Conversely, in presence of packet losses over a link, we randomly pick the current link RTT from the second half of the empirical RTT distribution.

As a final observation, the employed network model assumes that the additional network load due to the usage of multiple paths (instead of a single path) has negligible impact on network behavior (e.g. on the packet loss rate).

5.3.2 Edge Server Selection Policies

In conventional Web infrastructures (i.e. not leveraging parallel invocation and path diversity), client requests are routed toward a single edge server over

a single path, and the selected edge server is typically the one on the shortest path to the client. This mechanism may be straightforwardly adopted in our parallel invocation protocol by selecting the closest edge servers to the client, or one may envision the development of more sophisticated policies taking into account specific topological information in order to achieve larger benefits from the multi-path approach.

To cope with a relatively wide spectrum of possibilities, we implemented the following three selection policies in our simulator:

- *Shortest Paths.* Simply choose the closest edge servers to the client, employing hop counts as distance metric. In the following, we will refer this selection policy to as SP.

- *Disjointness Ordered Paths.* Always select the edge server on the shortest path. Then choose the edge servers whose paths to the client have a minimum number of links in common with the shortest path. If more than one server has the same number of joint links with the shortest path, choose the one having minimum length (measured in hop counts). In the following, we will refer this selection policy to as DP.

- *Disjointness×Length Ordered Paths.* Always select the edge server on the shortest path. Then choose the edge servers whose paths have the minimum values of the product between (i) the correlation with the shortest path and (ii) the additional length with respect to the shortest path. With this policy, if the path toward an edge server is highly disjoint from the shortest path, but such edge server is very far from the client, this edge server will not be considered by the client as a good candidate for the parallel invocation scheme. In the following, we will refer this selection policy to as D×LP.

5.3.3 Transactional Workload Model

In order to model the activities and resource consumption of the database servers we based our simulation model on the detailed distributed database system simulator first used in [55] to evaluate the performance of the PROMPT real-time distributed commit protocol and later employed in a number of works, e.g., [115, 57, 58, 129], addressing "DBMS-centric" topics ranging from advanced commit protocols to caching techniques for dynamically generated Web pages.

The original simulator, whose thorough description can be found in [56], is tailored for client/server distributed transaction processing environments where transaction coordinators are colocated with the database systems. Also, it provides a very simple network model representative of scenarios where involved processes are interconnected by a fast local area network. On the other

hand, it models very detailedly resource consumption due to network communication, data access and manipulation, and logging activities to support a variety of distributed commit protocol. Also, it ensures serializability of transaction execution by means of an accurate model of the Strict Two Phase Locking (S2PL) concurrency control mechanism [15].

The simulation components modeling database servers' resource consumption and concurrency control were extended and integrated with ad-hoc developed simulation software aimed at providing the following features:

- Integrate within the pre-existing S2PL concurrency control algorithm the MIP schemes so to allow differentiated treatment of sibling transactions. Data accesses among non-sibling transactions are instead regulated according to the original S2PL concurrency control scheme.
- Implement an accurate model of the database system's buffer management, so to better evaluate the resource consumption due to the execution of sibling transactions.
- Explicitly modeling the behavior of the different processes interacting in a three-tier system (i.e. clients, edge servers and data centers)
- Integrate the detailed network model described in Section 5.3.1.

For what concerns the transactional workload model used in the simulation, we exploited the so called "shopping workload", namely the reference transaction profile specified by TPC-W [120]. This benchmark is widely used for measuring the performance of e-Commerce systems, and relies on simulation of a breadth of activities of a business oriented transactional Web application. The shopping transaction profile is derived by TPC-W on the basis of the composition of two different customer profiles (also referred to as customer interactions) known as *browse* and *order*, respectively. The browse interaction involves browsing as well as querying activities, while the order interaction involves real update of data at the data centers. The shopping transaction profile is based on a composition of 80% browse interactions and 20% order interactions.

5.3.4 System Settings

For what concerns the size of the data set maintained at each data center and other system settings, we exploited the study in [71], where a global data set size of about 20 GB has been presented as a reasonable value for typical e-Commerce applications. In that study, the DBMS residing at the data center has 4 KB page size and is run on an IBM eServer xSeries 255 machine, with 4 CPUs (1.5 GHz), 8 GB of RAM storage, 12 IBM U320 disks (15000 RPM),

Topology	BRITE	NLANR
#nodes	5000	6474
#edges	5000	24467
Edge Servers Selection Policy	average path length between client and edge server	
SP	9.1	3.0
DP	9.3	3.1
D×LP	9.1	2.3
Edge Servers Selection Policy	average path length between edge server and data center	
SP/DP/D×LP	8.9	3.0
Edge Servers Selection Policy	average correlation ratio of used network paths (client side)	
SP	0.53	0.42
DP	0.46	0.35
D×LP	0.52	0.41

Table 5.1: Summary of Topological Parameters.

running Windows 2000 Advanced Server. Also, the DBMS is placed on a 5-disk hardware RAID-0. For this data set size, the characterization of the shopping transaction profile presented in [71] gives rise to an average number of 35 referenced pages for each interaction, with 96.6% of page references in read only mode, and 3.4% of page references in write mode. Resource consumption at the data centers while handling the interactions proper of the shopping transaction profile have been explicitly simulated in our analysis on the basis of such benchmarking results in [71].

We considered a service delivery infrastructure consisting of six back-end data centers and twenty edge servers. As shown in previous studies related to content delivery applications [7, 8], the number of paths that is expected to maximize the benefits from parallel invocation protocols is on the order of two. Hence we focused on the case of two edge servers contacted in parallel by the client. Fixed this setting, for the reader's convenience, we report in Table 5.1 a summary of the main parameters related to the different analyzed network topologies, together with information on the length and correlation of network paths for the different edge server selection policies (i.e. SP, DP and D×LP). These data have been obtained by considering clients spread in 500 different locations across the network.

In the simulation study we explicitly avoided to model caching of DBMS data at the edge servers. This choice derives from that, as outlined before, this type of caching requires explicit mechanisms for the maintenance of the consistency of replicated data [51], which might impact on the latency seen

by the users. Hence, we excluded caching of DBMS data in order to avoid any interference due to these mechanisms while performing the evaluation of the parallel invocation protocol. On the other hand, we remark that the absence of caching mechanisms at the edge servers yields to a worst case evaluation for the system saturation point since the back-end data centers need to manage the whole volume of interactions requested by the clients, including those interactions that would access data in read mode only (i.e. browse interactions). Given that the back-end data centers typically represent the system bottlenecks (especially when employing a an amount of edge servers relatively larger than the number of data centers within the infrastructure), our study focus on a representative situation for the evaluation of the effects of the overhead from our protocol on the system throughput. At the same time, we gathered statistical data by only considering the latency experienced by users really performing updates of application data, for which caching of DBMS data at the edge servers provides no advantage due to the fact that the corresponding requests are redirected to the origin data centers in order to manipulate the original data copy. This has been done to ensure fairness in the evaluation. Also, given that we do not consider the scenario in which the edge servers perform data caching, in our simulation study we do not to explicitly model resource consumption at the edge server side.

Finally, to capture network congestion/overload situations, we have set the parameter q of the Gilbert model to the value of 0.8, which corresponds to an expected burst loss length of 1.25. (It has been shown [125] that consecutive losses rarely last more than four packets and the value q=0.8 corresponds to the longest average burst length measurement we are aware of.) For what concerns the parameter p, we have considered two different values in the simulation study, selected as representative of interconnection between edge serves and data centers either via Internet or via a (virtual) private network under the control of the ASPs. In the former case, p was set to yield a moderate end-to-end loss rate of 5% for an average path length of 3 to 16 hops, depending on the topology. In the latter case, p was set to yield the extremely reduced end-to-end loss rate of 1% for the same average path lengths. The message size distribution has been obtained through a Pareto with α=1.5 and b=2.

5.3.5 Results

We report in Figure 5.6 and in Figure 5.7 the cumulative distribution function (CDF) of browser perceived response times for the BRITE and the NLANR topologies. In other words, we report on the Y-axis the experimentally evaluated probability for a browser to experience response time lower than the corresponding value on the X-axis. The plots report results for both a baseline protocol where clients interact with a single edge server and our parallel

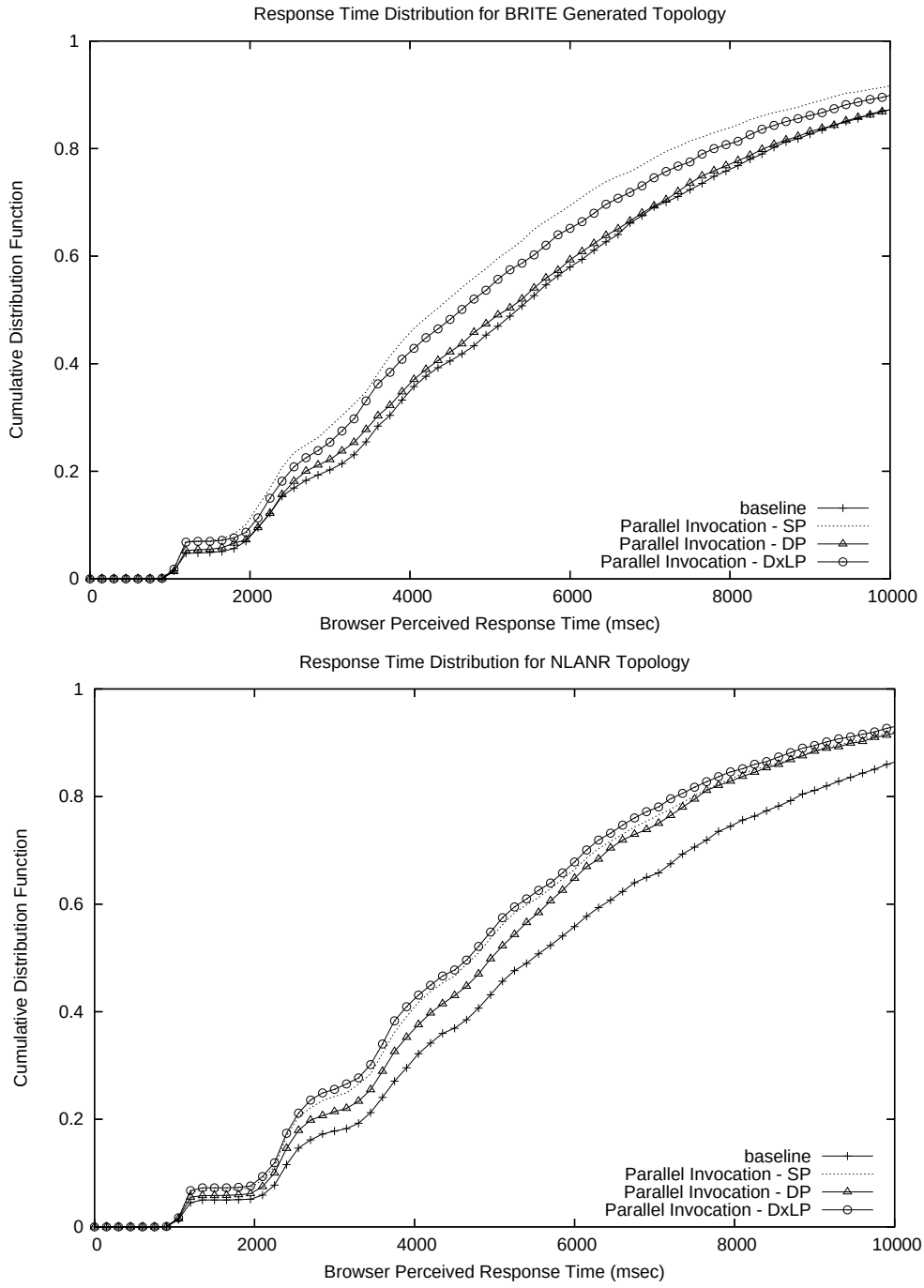


Figure 5.6: Browser Perceived Response Time CDF for the Case of Edge Servers and Data Centers Communicating via Internet.

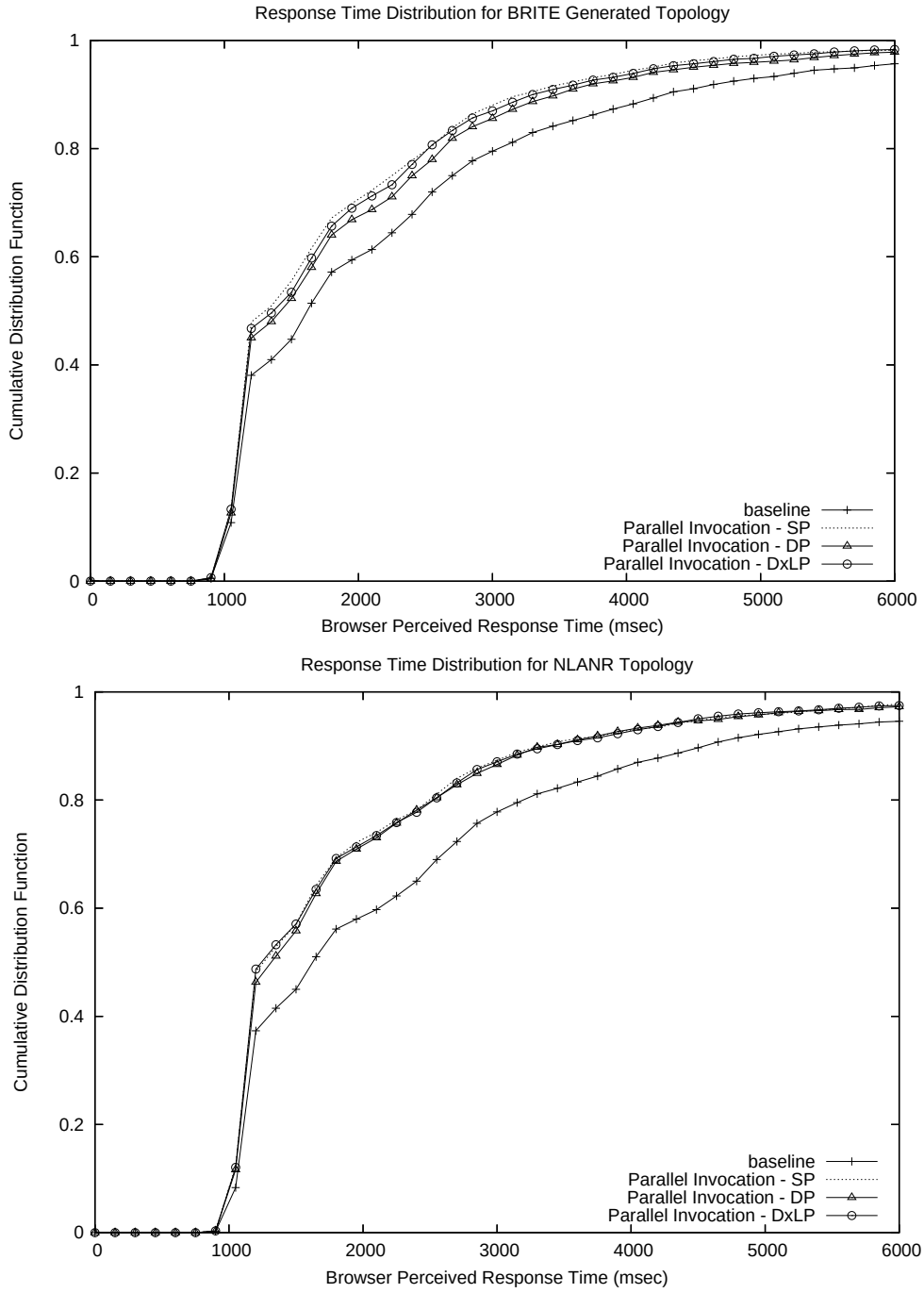


Figure 5.7: Browser Perceived Response Time CDF for the Case of Edge Servers and Data Centers Communicating via a (Virtual) Private Network.

invocation protocol (with the three different policies for selecting the edge servers to be contacted in parallel by the client).

By the plots we get that our parallel invocation protocol provides remarkable benefits, in terms of increased system responsiveness. For the case of edge servers communicating with data centers via the Internet (see Figure 5.6), our parallel invocation scheme allows achieving browser perceived response times less than 7 seconds (i.e. less than the maximum value complying with a reasonable expectation for an interactive end-user [130]) in about the 80% of the cases, whereas the baseline protocol achieves response times less than 7 seconds in about the 65% of the cases. An interesting, not so intuitively result highlighted by Figure 5.6 is that the DP edge server selection policy (which chooses the alternative path(s) in order to maximize path disjointness with respect to the shortest path) leads to a reduction of the benefits provided by the parallel invocation protocol. We have found that this depends on that the alternative path selected by DP might be significantly longer than the one selected by the other policies, excessively penalizing the performance of the request transmitted along that path.

The results related to the case of communication between edge servers and data centers via a (virtual) private network (see Figure 5.7) confirm the previous tendencies, with the only observation that, compared to the case of Internet based communication, this time we expect higher system responsiveness due to the more controlled network behavior at the side of the Web infrastructure (recall that for this configuration the parameter p has been set to obtain the extremely reduced packet loss rate of 1% over a path). Hence, the advantages from the parallel invocation protocol need to be evaluated for response time on the order of the reasonable value of 3/4 seconds, which is guaranteed by the multi-path protocol in about the 90% of the cases. Instead, even in such a controlled network scenario, the baseline protocol guarantees that response time value only in the 80% of the cases.

Overall, the plots show that our parallel invocation protocol determines (slightly) larger improvements of the perceived response times in the case of NLANR topology. This is coherent with the data reported in Figure 5.1 which points out that, with respect to the BRITE generated topology, the NLANR graph is characterized by a higher average degree of connectivity and a reduced correlation ratio among the network paths connecting the clients to the selected edge servers.

Another important observation from the plots is that they show remarkable benefits from the parallel invocation protocol even in case of no exploitation of path correlation information in the selection of the edge servers to be contacted in parallel by the client. In fact, the benefits achieved by users employing the correlation unaware selection scheme, namely SP, are in practice identical to those achievable with the other selection policies, or even larger. This is

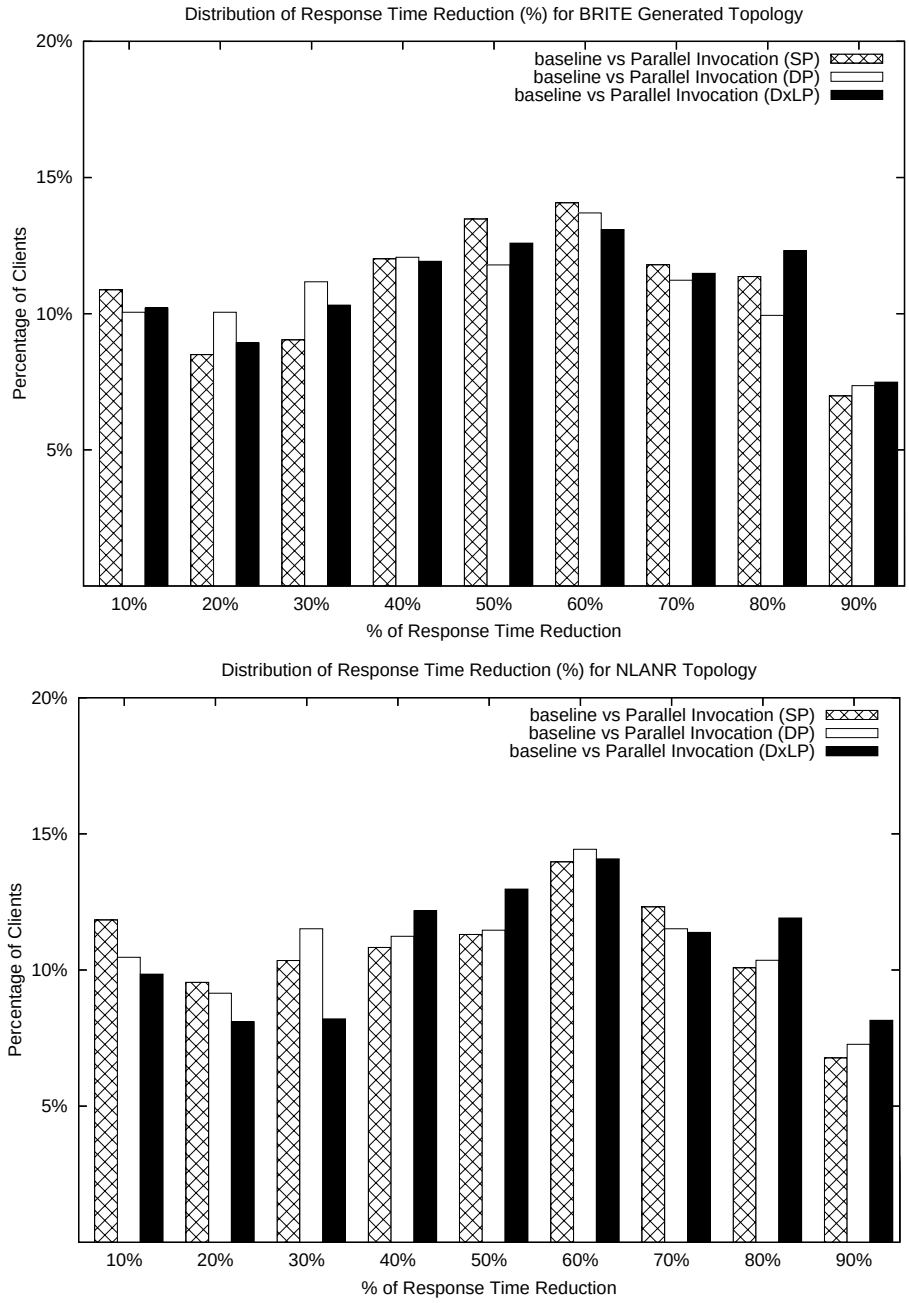


Figure 5.8: Distribution of Browser Perceived Response Time Reduction for the Case of Edge Servers and Data Centers Communicating via Internet.

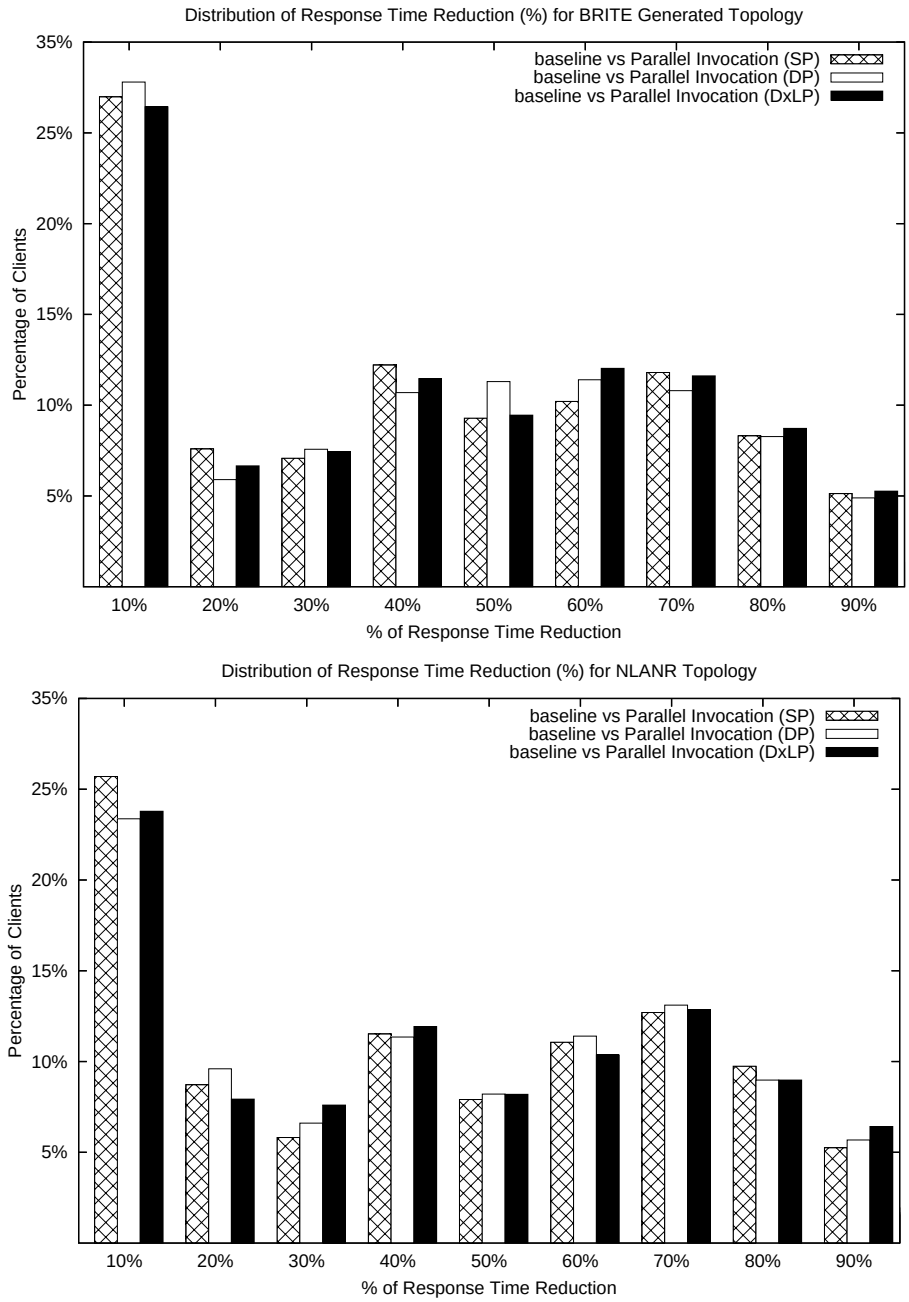


Figure 5.9: Distribution of Browser Perceived Response Time Reduction for the Case of Edge Servers and Data Centers Communicating via a (Virtual) Private Network.

an interesting result that confirms the feasibility of the parallel invocation protocol also in environments where it is difficult or impossible to infer the path correlation of the underlying network topology.

The plots in Figure 5.8 and in Figure 5.9 provide a different perspective to quantify the benefits achievable through the parallel invocation protocol. In these graphs we report the histograms of the percentage reduction in response time over the baseline for the two considered network topologies and for the three edge server selection policies SP, DP and D×LP. Such a data visualization highlights that there is a relevant percentage of clients experiencing a remarkable reduction in the perceived response time (evaluated as $\frac{Time_{baseline} - Time_{parallel\_invocation}}{Time_{baseline}}$) when the parallel invocation protocol is used. Specifically, in all the topologies at least the 50% of clients get response time reduction greater than (or equal to) 50%. Also, the 25% of clients get response time reduction of at least 70%.

5.3.6 On the Overhead

The previous plots allowed us to evaluate the effectiveness of our protocol from the point of view of an interactive end-user. However, the effectiveness of any solution in support of a transactional application can not be realistically evaluated without taking into account its impact in terms of server side throughput.

The parallel execution of multiple instances of a same distributed transaction obviously adds some overhead with respect to a baseline approach in which clients' requests are serially submitted to a single application server. In current infrastructures for Web applications, the trend is to have a very high degree of replication of the edge servers, hence the system bottlenecks tend to be the back-end data centers, which need to support the whole volume of multiple transaction instances associated with a same request from whichever geographically dispersed client. As a consequence, the overhead at the data centers imposed by the parallel invocation protocol is representative of the possible impact of the protocol on the throughput and saturation point of the whole infrastructure.

To this purpose, we report in Figure 5.10 the plots related to the variation of the system response time (and so of the system saturation point) while varying the workload in terms of number of concurrently active browsers (recall that according to the TCP-W specification, in our simulation each simulated browser requests interactions according to an exponentially distributed think time interval with mean value 7.5 seconds). These plots have been obtained considering the NLANR network topology, and the scenario of edge servers and data centers communicating via Internet. However, very similar results were obtained also when considering the BRITE generated topology and the

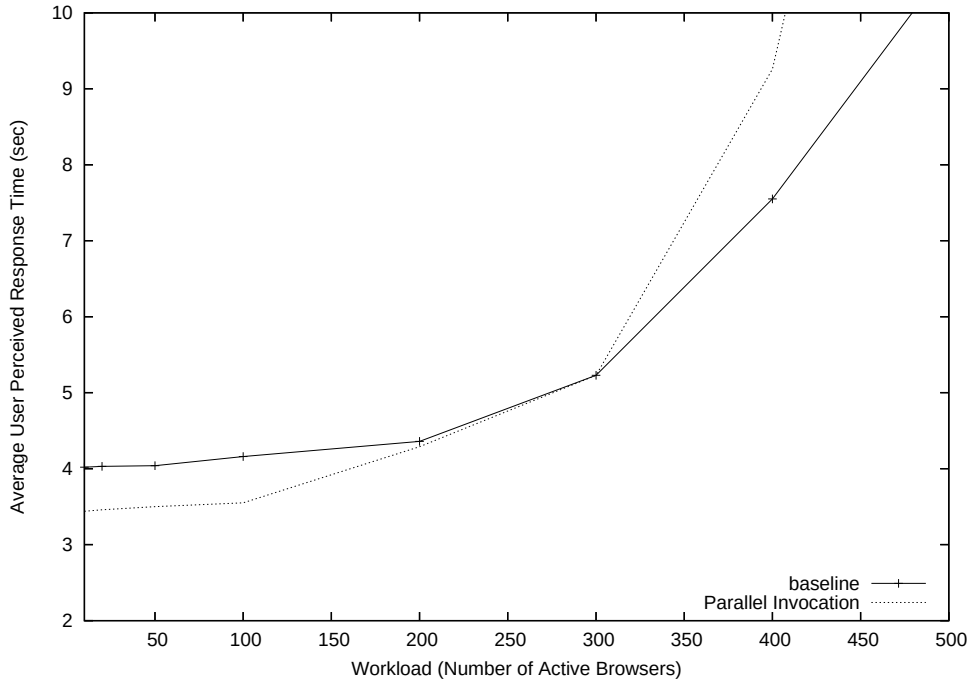


Figure 5.10: System Saturation Point.

scenario of edge server and data centers communicating via a (Virtual) Private Network, which is the reason why we avoid reporting them here.

By the plots we observe that our parallel invocation protocol produces only a limited reduction of the system throughput (i.e. of the system saturation point). Specifically, it gives rise to system saturation when the workload is on the order of about the 85% of the workload that gives rise to saturation when the baseline is employed. We found that the relatively low overhead imposed by our protocol, and consequently the small reduction in the database throughput, is mainly due to that sibling transactions normally access the very same data set. This means that some of the costs for accessing that data set (e.g. the cost for transferring the data set from disk to the database buffers in volatile memory) are paid only once.

Chapter 6

Specialization of the Results for the Single Back-end Database Scenario

The e-Transaction protocols presented in the previous chapters address the general case of transactions involving an arbitrary number of distributed, autonomous back-end databases. This naturally implies that those solutions could be adopted even in the scenario in which the application interacts with a single centralized back-end database server. However, these solutions have been natively designed to cope with atomic distributed transactions. Hence they result structurally dependent on a two-phase commit protocol, which is a source of unnecessary overhead (in terms of additional messaging and database logging activities) when dealing with applications accessing a single database, where a cheaper one-phase commit protocol can be safely employed. Given that such a scenario is in practice commonly found in a wide range of diverse application domains [43], it would be highly desirable to design e-Transaction protocols specifically optimized for this case.

This is precisely what we do in this chapter, where we present an e-Transaction protocol specialized for the scenario of applications interacting with a single back-end database. This protocol borrows some of the key ideas already introduced in the previous chapters and adapts them so to maximally exploit the implications related to the adoption of a one-phase commit protocol. It is worthy underlining that the solution presented in this chapter retains the remarkable theoretical and practical benefits characterizing the e-Transaction protocol presented in Chapter 4. Specifically, not relying on any coordination scheme among application server replicas, it achieves very high scalability levels and very limited overhead. Further, by avoiding forced termination of previously issued requests, it preserves its correctness even in a

pure asynchronous system without relying on any assumption on the accuracy of the failure detection mechanism. Our protocol improves the state of the art exactly in that none of the existing solutions, not even those explicitly optimized to tackle the single back-end database scenario, exhibit both of these desirable features together. Additionally, the avoidance of the termination scheme allows this solution to effectively support parallel invocation of transactional requests, thus also permitting the treatment of performance failures in a manner specifically optimized for the single back-end database scenario.

Beyond presenting the protocol, we also provide a formal proof of its correctness. Next, analogously to what we did for our previous solutions, we discuss practical issues and possible integration with conventional software technology. We conclude this chapter by comparatively evaluating the protocol with state of the art approaches.

6.1 Model of the System

For what concerns client and application server processes, we consider here the same identical model used in Chapter 4 and Chapter 5. Hence, we avoid re-describing them and move directly to present the database server model.

6.1.1 Database Server

We assume that the database server supports custom transaction demarcation and concurrency control schemes, which are almost identical the ones proposed in Section 4.1.1 and Section 4.1.1 to support MIP. (Such an assumption will be later relaxed in section 6.5, where we will show how to pragmatically implement the protocol by relying exclusively on standard transactional technology.) Hence, also in this case we assume that the concurrency control scheme avoids that sibling transactions incur mutual blocking due to conflicts on data items accessed in common. The fundamental difference with the concurrency control scheme underlying MIP is that, here, we are not faced with avoiding transaction conflicts due to the presence of pre-commit locks (which only arise in distributed transaction processing scenarios where an atomic commit protocol, e.g. 2PC, has to be employed). More formally, we assume that in case a transaction T requires (read/write) access to some data item d previously accessed (written/read) by a not yet committed (i.e. pending) transaction T', T is granted access to the pre-image of d with respect to the execution of T' if T and T' share the same req_id (i.e. they are sibling transactions). Hence any update performed by a not yet committed transaction T' is not visible to any sibling transaction T. No assumption is made on how concurrency control regulates data accesses of non-sibling transactions.

We model with the `decide` primitive, the database server interface for invoking the commitment/abort of a pending transaction. `decide` takes in input a XID (i.e. a request identifier and a transaction instance identifier) and returns commit/rollback depending on the final decision the database takes for the transaction. Also, as in conventional database technology, if the database server crashes while processing a transaction, it does not recognize that transaction as an active one after recovery. Therefore, if the `decide` primitive is invoked with an identifier associated with an unrecognized transaction in input, then the return value of this primitive is rollback. We assume the `decide` primitive is non-blocking, i.e. it eventually returns unless the database server crashes after the invocation.

The database server maintains some recovery information, which we refer to as ITP (Information on Transaction Processing) in analogy to Chapter 3. The ITP is used to determine whether a sibling transaction has already been committed and, in that case, to retrieve the non-deterministic result produced by the execution of that transaction. However, with respect to the protocol in Chapter 3, both the ITP structure and the logic that manipulates it are simplified. More specifically, the ITP consists of (i) the identifier of the transaction and (ii) the transaction result (i.e. the output of the `compute` primitive executed by the application server that committed the transaction).

The ITP is manipulated by means of the `insert` and `lookup` primitives. `insert` takes a client request identifier and a result as input parameters, and records them (i.e. inserts the corresponding tuple) within a dedicated database table, which we refer to as ITP table. As in Chapter 3, we assume the request identifier to be a primary key for that database table. Therefore, any attempt to insert the previous tuple within the database multiple times is rejected by the database itself, which is able to notify the rejection event by raising the `ITPDuplicatePrimaryKeyException` ([1]). In the following we say that a transaction whose commitment would imply a violation of the primary key constraint on the ITP table is *illegal*. Otherwise, we say that the transaction is *legal*.

A fundamental difference with the ITP manipulation logic prescribed in Chapter 3 is that, here, the `insert` primitive is executed within the same transactional context activated by the application server by means of the `compute` primitive (i.e. the ITP entry is stored in the database as part of the execution of the transaction initiated by the application server). Hence, differently from the solution in Chapter 3, in this approach no additional eager log operation is required at the database side to support the ITP manipulation, achieving a further reduction in the corresponding overhead.

[1] As we will see, the presence of an ITP entry associated with a client request identifier implies that a transaction associated with that client request has already been committed by the database.

For presentation simplicity, but with no loss of generality, we assume that the database server raises the `ITPDuplicatePrimaryKeyException` only at the time the `decide` primitive is invoked to commit the illegal transaction. In practical settings, however, our protocol could be employed also in case the database server were to validate the legality of the `insert` statement immediately, rather than at commit time. Also, as it normally happens in classical transactional processing technology, we assume that if the database detects that a transaction is attempting to violate a primary key constraint on the ITP table, it autonomously aborts that transaction.

The `lookup` primitive takes in input a client request identifier and returns the result stored within the corresponding entry in the ITP table, if any.

Both the `lookup` and the `insert` primitive are assumed to be non-blocking.

6.2 The Protocol

6.2.1 Client Behavior

The client behaves exactly like in our protocol presented in Chapter 4. Its pseudo-code is presented in Figure 4.3 and is not replicated here. Essentially the client keeps on retransmitting its request to different application server replicas on a timeout basis. Each request message is tagged with the same request identifier *req_id* and with a different, unique instance identifier *inst_id*. The client stops retransmitting requests and delivers a result as soon as the first Outcome message reporting a *commit* indication is received from an application server.

6.2.2 Application Server Behavior

The application server behavior is shown in Figure 6.1. Upon receipt of a Request message from the client, the application servers invokes the `compute` primitive to activate the business logic executing a transaction against the back-end database which is left pending (i.e. uncommitted). Then it asks the database to commit the transaction by sending out a Commit message, along with the result associated with the transaction execution returned by the `compute` primitive. Next the application server waits for a CommitACK message from the database server, reporting an outcome in the domain *commit*/*abort*, reflecting whether the database server was able to commit the transaction, and a result. As we will see in the next section the result specified within the CommitACK message could actually be different from the one previously calculated through the `compute` primitive and sent along with the Commit message: in such a case, the result contained within the CommitACK message is the one associated with the first sibling transaction to have been

committed by the database server for a given client request. If the application server does not receive any CommitACK message within a timeout period, the application server simply keeps on retransmitting the Commit message. As soon as a CommitACK message is received, the application server sends in its turn an Outcome message to the client specifying the outcome and the result returned by the database through the CommitACK message.

```
Class ApplicationServer{
 Outcome outcome;
 Result result;

 void main(){
   while(true){
     wait receive Request[req, < req_id, inst_id >] from client or from DB;
     result =compute(req, < req_id, inst_id >);
     repeat {
       send Commit[< req_id, inst_id >, result] to DB;
       wait (receive CommitACK[< req_id, inst_id >, outcome, result] from DB)
            or (TIMEOUT);
     } until received CommitACK[< req_id, inst_id >, outcome, result] from DB;
     if (Request was received from client)
       send Outcome[< req_id, − >, outcome, result] to client
   } // end while true
 } // end main
} // end class
```

Figure 6.1: Application Server Behavior.

6.2.3 Database Server Behavior

The database server behavior is formalized by means of the pseudo-code in Figure 6.2. This server executes one task triggered by the receipt of the Commit message, and a background task aiming at ensuring that every computed transaction is eventually aborted or committed.

Task 1 : Upon the receipt of a Commit message for a transaction having $XID =< req\_id, inst\_id >$ and specifying a corresponding *result* value, the database server attempts to insert the specified result within the ITP entry associated with the *req_id* request identifier. Next, the `decide` primitive is invoked to attempt to commit that transaction. If the transaction is successfully committed, this means that any subsequent attempt to commit a sibling transaction (i.e. a transaction associated with the same request identifier) will be rejected by the database by raising an `ITPDuplicatePrimaryKeyException`. This is indeed the key mechanism we adopt to avoid duplicate transaction executions. In such

```
Class DatabaseServer{
 List ASlist = {AS_1, ..., AS_n};
 ApplicationServer AS;
 Result result;
 Outcome outcome;
 InstanceIdentifier inst_id;
 on stable storage Counter counter = InitialValue;

 void main(){
   while(true){
     cobegin
     || wait receive Commit[< req_id, inst_id >, result] from AS_i // Task 1
       try {
         insert(req_id, result);
         outcome= decide(< req_id, inst_id >, commit);
         if (outcome==abort) result=nil;
       }
       catch (ITPDuplicatePrimaryKeyException ex) {
         result =lookup(inst_id);
         outcome=commit;
       }
       send CommitACK[< req_id, inst_id >, outcome, result] to AS_i;
       if (outcome==commit) {
           for every pending transaction having XID =< req_id, − > do
             decide(< req_id, − >, abort);
       } //end if
     || background: // Task 2
       for every transaction having XID =< req_id, − > computed with request content
       req as input longer than TIMEOUT period {
         AS = ASlist.next();
         inst_id =< GetMyCategory(), + + counter >;
         send Request[req, < req_id, inst_id >] to AS;
         reset TIMEOUT period for the transaction having identifier XID;
       } // end for every
   } // end while
 } // end main
} // end class
```

Figure 6.2: Database Server Behavior

a case, the database server sends back to the application server a CommitACK message carrying a *commit* outcome as well as the original *result* specified by the application server within the Commit message.

If the `decide` primitive fails to commit the transaction, there are two cases to consider. If the transaction were aborted because it attempted to duplicate a primary key on the ITP table ([2]), the `lookup` primitive is invoked to retrieve the result stored by the sibling transaction that previously committed. Then a CommitACK message is sent back to the application server specifying the *commit* outcome and carrying the result just retrieved from the ITP table. Note that in this case the database server returns *commit* even though the transaction previously activated by the application server through the `compute` primitive was actually aborted. This is safe, since the application server is provided with the (non-deterministic) result value produced by the execution of the first, and only, sibling transaction to have been committed by the database.

The only case left is the one in which the `decide` primitive returns an *abort* outcome without raising the `ITPDuplicatePrimaryKeyException`, as in the case of unilateral aborts due to, e.g., the concurrency control mechanism. In this case the database returns a CommitACK message reporting an *abort* outcome and a *nil* result value.

Prior to completing execution of **Task 1**, if a *commit* outcome was sent back along with the CommitACK message ([3]), the database makes sure that any pending transaction associated with the specified client *req_id* value is aborted. This is required to ensure that any previously computed transaction gets eventually aborted. Note that aborting pending transactions associated with previously issued request instances takes place only after having ensured that one, among those sibling transactions, was already committed. This preserves the liveness of the end-to-end interaction even in those scenarios where no accuracy at all on the failure detection can be guaranteed, like in a pure asynchronous system.

Task 2: The database server retransmits, tagged with a monotonically increasing instance identifier, the request associated with each transaction that remains pending for longer than a timeout period. This is done to ensure that any transaction that was computed but left uncommitted due to a crash of the corresponding application server gets eventually committed or aborted. To allow such a behavior, we assume that the

[2] In this case the execution jumps at the first statement within the catch clause.

[3] Meaning that either the transaction specified in the Commit message could be committed or that a sibling transaction was already found to have committed.

database can store the request content, upon activation of the transaction, in volatile memory (note that this is not shown in the database pseudo-code as we do not explicitly model the `compute` phase). In case of crash, in fact, any pending transaction would get anyway aborted.

6.3 Proof of Correctness

In this section we provide a formal proof of the protocol correctness with respect to the e-Transaction properties. For the case of single back-end database, these properties have been formalized in [43], and are recalled below:

Termination:

T.1 If a client issues a request, then unless it crashes, it eventually delivers a result.

T.2 If an application server computes a result, then the database commits or aborts the corresponding transaction.

Agreement:

A.1 No result is delivered by the client unless the database commits the corresponding transaction.

A.2 The database does not commit more than a single transaction for each request.

Validity:

V If the client delivers a result, then the result must have been computed by an application server with, as a parameter, a request issued by the client.

6.3.1 Correctness Assumptions

The correctness of the protocol essentially relies on the same set of assumptions also required by the protocol we presented in Chapter 4, which we briefly recall here for reader's convenience.

We assume a pure asynchronous system, hence not requiring any accuracy from the underlying failure detection scheme, which is in fact implemented in our protocol by means of plain timeouts, i.e. an arbitrarily inaccurate failure detection mechanism in an asynchronous system.

Also, we assume that at least an application server is correct and that the database server is *good*, which means that: (1) it always recovers after crashes, and eventually stops crashing (i.e. it eventually becomes correct), and (2) if the application servers keep re-trying a *legal* transaction, the database can eventually commit it. Note that the assumption on database goodness expressed in point (2) has been rephrased in this chapter so to require that the

database is eventually be able to commit a *legal* transaction, i.e. a transaction that does not violate an ITP primary key constraint. This is required in order to reflect the fact that in this protocol we have forced an explicit primary key conflict among the transactions associated with the same client request, implying that the database will not be able to commit more that one of these transactions.

6.3.2 Correctness Proof

Lemma 6.3.1 *If a correct application server sends a* Commit *message to the database server for a transaction, the application server eventually receives a* CommitACK *message for that transaction.*

Proof (By Contradiction) Assume by contradiction that a correct application server sends a Commit message to the database server and that it never receives a CommitACK for the corresponding transaction.

In this case, the correct application server keeps on retransmitting the Commit message to the database server indefinitely. Hence, a Commit message will be sent by the application server to the database server at time $t' > t$, where t is the time after which the database server stops crashing and remains up. Given that after time t both the correct application server and the database server are always up, for the assumption on the reliability of the communication channels we can claim that the database server will eventually receive the Commit message. Also, the database server will eventually take a decision through the `decide` primitive (since it does not crash anymore and `decide` is non-blocking) and will send a CommitACK message to the application server. Again, since communication channels are assumed to be reliable, the correct application server will eventually receive that CommitACK message. Therefore the assumption is contradicted and the claim follows. $Q.E.D.$

Lemma 6.3.2 *If a correct application server keeps on receiving* Request *messages tagged with the same request identifier and different transaction instance identifiers, it eventually receives a* CommitACK *message tagged with that request identifier and carrying a commit outcome.*

Proof (By Contradiction) Assume by contradiction that a correct application server receives an unbounded number of Request messages tagged with the same request identifier and different transaction instance identifiers and never receives a CommitACK specifying a *commit* outcome for a corresponding transaction.

If a correct application server receives a Request message, it calls the primitive `compute` which, being non-blocking, eventually returns. Therefore the

application server eventually sends to the database server the Commit message. By Lemma 6.3.1, a CommitACK message for the transaction is eventually sent back by the database server and is eventually received by the correct application server. Two cases are possible:

1. If the CommitACK message received from the database server carries a *commit* indication, meaning that the database either successfully committed the specified transaction or that the specified transaction was illegal given that a sibling transaction had been already committed, then the assumption is contradicted and the claim follows.

2. If the CommitACK message received from the database server carries an *abort* indication, it means that the transaction was legal (given that it would have otherwise risen an `ITPDuplicatePrimaryKeyException` and we would have fallen in case 1) but that the database server was unable to commit the required operations due, e.g., to decisions of the concurrency control mechanism.

We note anyway that case 2 (i.e. the only one that does not contradict the assumption) can't occur indefinitely as the correct application server keeps indefinitely retrying transactions and we have assumed that there is a time after which the database server remains up and is able to commit any legal transaction. Hence the assumption is contradicted and the claim follows. *Q.E.D.*

Termination T.1 - *If a client issues a request, then unless it crashes, it eventually delivers a result.*

Proof (By Contradiction) Assume by contradiction that the client issues a request, does not crash and does not eventually deliver a result, meaning that it never receives an Outcome message with a *commit* indication.

In this case, the client keeps on retransmitting the Request message tagged with the same request identifier *req_id* and different transaction instance identifiers *inst_id* to the application servers indefinitely. As we have assumed that channels are reliable and that at least an application server is correct (i.e. it does not crash), an unbounded number of Request messages will eventually be delivered to a correct application server. By Lemma 6.3.2, it follows that the correct application server will eventually receive a CommitACK message tagged with that request identifier and carrying a *commit* indication. In this case the correct application server sends back to the client an Outcome message with a *commit* indication and, given that client and application server do not crash

and that channels are reliable, this message will eventually be received by the client. Hence, the assumption is contradicted and the claim follows. *Q.E.D.*

Termination T.2 - *If an application server computes a result, then the database commits or aborts the corresponding transaction.*

Proof (By Contradiction) Assume by contradiction that the database server computes a result associated with the $XID =< req\_id, inst\_id >$ transaction identifier and does not eventually commit or abort the corresponding transaction, meaning that the `decide` primitive is never invoked with the XID transaction identifier as an input parameter and that this transaction remains pending for ever. This also implies that the database server does not crash after having computed this transaction, otherwise the transaction would have been aborted.

In this case, the database server activates indefinitely **Task 4**, meaning that it keeps on indefinitely retransmitting to the application servers a Request message containing the request content which was passed in input when the pending transaction was activated through the `compute` primitive, and tagged with the same client request identifier and different instance identifiers. As we have assumed that channels are reliable and that at least an application server is correct (i.e. it does not crash), an unbounded number of Request messages will eventually be delivered to the correct application server.

By Lemma 6.3.2, it follows that the correct application will eventually receive a CommitACK message tagged with the *req_id* request identifier and carrying a *commit* indication. By channel reliability, this implies that the database server must have previously sent that CommitACK message. At this point there are only two possibilities:

1. The database server crashes immediately after the transmission of the CommitACK message to the application server, in which case the transaction with identifier XID is aborted. Hence the assumption is contradicted and the claim follows.

2. The database server, immediately after the transmission of the CommitACK message finds the local *outcome* variable set to the *commit* value. In this case it invokes the `decide` primitive passing as input parameters the *abort* outcome value and the transaction identifiers of every computed, still pending transaction having the request identifier *req_id*, including also the transaction identifier XID. Hence the assumption is contradicted and the claim follows.

Q.E.D.

Agreement A.1 - *No result is delivered by the client unless the database commits the corresponding transaction.*

Proof The client delivers a result only after it receives from an application server an Outcome message specifying a *commit* indication and tagged with the request identifier (*req_id*) corresponding to its previously issued request. Such a message is sent to the client if, and only if, the application server has received from the database server a CommitACK message with the *commit* indication for that request identifier. Also, the result contained in the Outcome message sent by the application server to the client, is the one specified in the CommitACK message.

A CommitACK message carrying the *commit* indication and the *req_id* request identifier is sent by the database server only after it has invoked the `decide` primitive for a transaction having request identifier *req_id*. This can give rise to two cases:

(1) The database server successfully commits the transaction associated to the client request (also successfully performing the insertion of the corresponding ITP tuple). In such a case the client trivially follows.

(B) The database server detects a primary key conflict due to the insertion of the ITP tuple. In this case, a transaction having the same *req_id*, and hence associated with a different instance of the same client request must have been already committed by the database, and the corresponding result must have been stored in the associated ITP tuple. Given that in this case, the database returns to the application server the result it retrieves from the corresponding ITP entry, the claim follows.

Q.E.D.

Agreement A.2 - *The back-end database does not commit more than one transaction for each client request.*

Proof (By Contradiction) Given the structure of the protocol, it is possible that multiple transactions associated with the same client request are activated by the application servers. Assume, by contradiction, that a generic number $N > 1$ of them are eventually committed.

In this case, the database server must have invoked multiple times the `decide` primitive for transactions associated with the same client request identifier. By the database server pseudo-code, this server invokes the `decide` primitive only after that the insertion of the unique client request identifier

together with the result of the data manipulation through the `insert` primitive has occurred. As a consequence, in order for the $N > 1$ transactions associated with the same client request to be committed, they must have performed a successful insertion of the unique request identifier within the database. However, this is impossible since the database maintains a primary key constraint on the request identifier, hence no more than one of those N transactions can successfully perform that insertion, whereas all the remaining ones will be aborted by rising an `ITPDuplicatePrimaryKeyException`. Therefore the assumption is contradicted and the claim follows. *Q.E.D.*

Validity V - *If the client delivers a result, then the result must have been computed by an application server with, as a parameter, a request issued by the client.*

Proof The client delivers a result only after it receives from an application server an Outcome message specifying a *commit* indication and tagged with the client request identifier (*req_id*) corresponding to its previously issued request. Such a message is sent to the client if, and only if, the application server has received from the database server a CommitACK message with the *commit* indication for that request identifier. This happens only if the result has already been committed, and a result associated with that request identifier can get committed only after an application server has computed it. An application server computes a result only after it has received the Request message with that request identifier either from the client, in which case the claim trivially follows, or from the database server. In the latter case, however, given that the database server does not spontaneously issue requests, it means that an application server must have previously received a Request message with that same request identifier from the client. Also in this case, therefore, the claim follows. *Q.E.D.*

6.4 Supporting Parallel Invocation

In this protocol, just like in the one presented in Chapter 4, fail-over is performed via a simple timeout based request retransmission logic and not via extermination of previously submitted requests. Hence also this solution lends itself to easily transmute into a parallel protocol analogous to the one described in Chapter 5.

In this case the modification to apply to the original pseudo-code of the various processes would be really minimal, requiring exclusively an obvious transformation in the client pseudo-code along the lines of the one shown in

Section 5.1.1. Conversely, the application and database server pseudo-codes could be seamlessly employed within the parallel invocation scheme without requiring any modification at all.

Finally, for what concerns the benefits, in terms of increased system responsiveness and (performance) failure masking, and the costs, in terms of additional resource consumption at the database server, our simulation studies in [100] show very similar results to the ones already reported in Section 5.3 hence we avoid reporting them here.

6.5 On Practical Issues

In order to allow the garbage collection of unneeded recovery information from the ITP database table, one could rely on the same mechanisms presented while describing our protocol in Chapter 4. Essentially, in the most likely scenario in which neither the client nor the databases experience timeout expirations, the database servers should piggyback on the CommitACK message directed to the application server an additional flag conveying information on that it did not perform any request retransmission associated with that same request identifier (and that it won't perform any request retransmission in the future). Also, the client should send an explicit acknowledgment message to the application server upon receipt of the Outcome message. At this point, the application server could simply notify the database to discard the corresponding ITP entry.

In case some request retransmission had to be performed either by the client or by the database server, the above acknowledgment scheme could be modified in the following way. The client and the database servers could inform the application server via a different acknowledgment message that retransmissions have already taken place. The application server could then ask the database servers to remove the transaction result from the corresponding ITP entry, while maintaining the request identifier. This would allow a considerable reduction of the storage space occupied by the recovery information.

Finally, just like in our previously presented protocols, an alternative, or possibly complementary, approach to garbage collection of recovery information, would be to rely on appropriately selected TTL values, after which deletion of the ITP entry from the database can be performed.

A pragmatic optimization of the protocol behavior, which we will refer to as "preventive check optimization", is based on the observation that when an application server receives a request retransmission, it is possible that some previous request instance has already been committed by the database. In the above version of our protocol however such an event is detected by the database server only after the transactional business logic has been completely

executed. We argue that, in order to reduce the user perceived fail-over latency, a simple optimization could be to tag the request retransmissions with an additional flag *check*, not included in the first request instance transmitted by the client. This would inform the application server that it is currently processing a retransmission of that request, so that it could specialize its treatment. Specifically, the application server could exploit the `lookup` primitive to discover whether some sibling transaction was already committed and, in the positive case, return the corresponding result by saving time and resources since neither the `compute` nor the `insert` would be executed. It is worthy underlining that such an optimization aims exclusively at pursuing performance benefits, as the safety properties of our protocol would still be ensured by exploiting the above described primary-key conflict mechanism among sibling transactions.

As hinted, a key feature of the protocol is to allow sibling transactions to execute in parallel, rather than relying on an extermination based approach to handle failure suspicions. Such an approach assumes that the database's concurrency control scheme is able to ensure the (eventual) progress of concurrently active sibling transactions despite the possibility of mutual data conflicts. A relevant pragmatic implication of dealing with a single back-end database is that, in this context, the concurrency control does not need to maintain durable (i.e. not unilaterally releasable) locks on data items. (These are instead required in the context of distributed transaction processing, during the pre-commit phase, in order to provide persistent guarantees on the ability to subsequently commit a transaction [17, 52, 117].) Given that in this case we are not concerned with durable locks, this protocol, differently from the one presented in Chapter 4, does not necessarily require to be supported by a custom, ad-hoc concurrency control mechanism. Specifically, we note that a classic optimistic concurrency control (OCC) (in which transactions do not mutually block and are validated at commit time) would be a perfect candidate to support the protocol, providing the potentialities for real parallelism, which could even allow maximal exploitation of the benefits provided by the parallel invocation approach discussed in Section 6.4. On the other hand, if the database server adopted a pessimistic concurrency control (PCC), e.g. a strict two-phase locking [17, 52], the transactions associated with a given request could be blocked by a pending sibling transaction instance, until this latter either commits/aborts or experiences lock timeout for deadlock detection at the back-end database. The most critical scenario with PCC is probably the one in which the application server processing the "blocking" transaction were to crash before completing it. Even in this case, anyway, given that the database must not maintain any durable lock, the blocking transaction would eventually be aborted by the timeout based deadlock detection mechanism. This prevents blocking scenarios of unbounded duration and ensures,

in practical contexts, that the liveness of the end-to-end interaction does not get compromised. On the other hand, given that PCC could reduce the parallelism level among sibling transactions, it could reveal inviable in case of parallel invocation aimed at masking performance failures.

Another practical issue is related to that our protocol assumes that a committing transaction can explicitly abort any pending sibling transaction so to ensure that no transaction remains uncommitted for ever. Such a behavior is directly supported by the standard XA transaction demarcation API. However, the XA APIs are normally employed in distributed transaction processing environments and may not be supported by DBMS products that are not designed to participate in distributed transactions, which normally provide only a JDBC/ODBC connection oriented programming interface. In these environments, it is unfortunately not possible to request the abort of a transaction associated with a different thread of control, i.e. with a different database connection. However, in practical settings, it is the DBMS itself that eventually aborts in an unilateral way any transaction that is left pending for too long in order to release valuable system resources. One could therefore rely on the DBMS to ensure the termination of pending sibling transactions, and develop a pragmatical implementation of our protocol based exclusively on universally supported JDBC/ODBC. This standard feature of real DBMSs could be exploited also to avoid reliance on the retransmission logic activated by the database server's background task (see **Task 2**) to ensure eventual termination of pending transactions. This could further simplify the implementation of our protocol and its integration with COTS database products.

For what concerns the implementation of the ITP manipulation logic (specifically the insertion of the ITP entry within a database table and the handling of the corresponding duplicate primary key exception) one can envision a number of viable alternatives. The simplest one would probably be to explicitly perform the ITP manipulations at the application level, by explicitly extending the transactional business logic. A more elegant approach would be to delegate this task either the JDBC/ODBC driver or, in the case, of applications relying on container managed transactions [114], to the container module providing application components with lower level transactional services, e.g. OTS in the J2EE platform. This would permit to transparently enforce the e-Transaction guarantees without requiring modifications to the application level logic.

As a final additional note, the same methodology discussed in Section 4.4 could be adopted also in this case, so to allow clients to correctly recover after crashes.

6.6 Comparison with State of the Art Approaches

In this section, we perform a qualitative and quantitative comparison between our protocol and the existing e-Transaction solution natively optimized to tackle the single back-end database scenario. Such a protocol, which we refer to as FG in the following, was presented by Frolund and Guerrauoi in [43]. Then, we present experimental data allowing us to evaluate the overhead introduced by our solution with respect to a baseline protocol which does not provide any reliability guarantee. These data help us to quantify the inherent cost of reliability exhibited by our solution in a realistic transaction processing environment.

6.6.1 Qualitative Comparison with FG

Like our solution, the FG protocol also avoids coordination schemes among application server replicas and relies on the idea of logging some recovery information at the back-end database while processing the transaction.

During normal operation mode, both our protocol and FG actually exhibit the same behavior, shown in Figure 6.3.a. However, differently from our proposal, the FG protocol handles failure suspicions through a termination phase executed upon timeout expiration at the client side, see Figure 6.3.b. During this phase, the client sends, on a timeout basis, terminate messages to the application servers in the attempt to discover whether the transaction associated with the last issued request was actually committed. An application server that receives a terminate message from the client tries to rollback the corresponding transaction, in case it were still uncommitted (possibly due to crash of the application server taking care of it). At this point the application server determines whether the transaction was already committed by testing the presence of the recovery information. In the positive case, the application server retrieves the transaction result to be sent to the client. Otherwise, a rollback indication is returned to the client in order to allow it to safely send a new request message (with a different identifier) to whichever application server.

Our protocol avoids the termination phase since it makes request retransmissions idempotent operations thanks to the use of a primary key constraint imposed on the recovery information supporting the testable transaction abstraction. From the point of view of performance, the avoidance of the termination phase reduces the latency of the fail-over phase as compared to [43], as it will be clearly quantified in Section 6.6.2. More importantly, avoiding the termination phase makes our protocol a more general solution. In fact, by admission of the same authors, the employment of such a phase limits the usability of their protocol to environments where it can be ensured that a

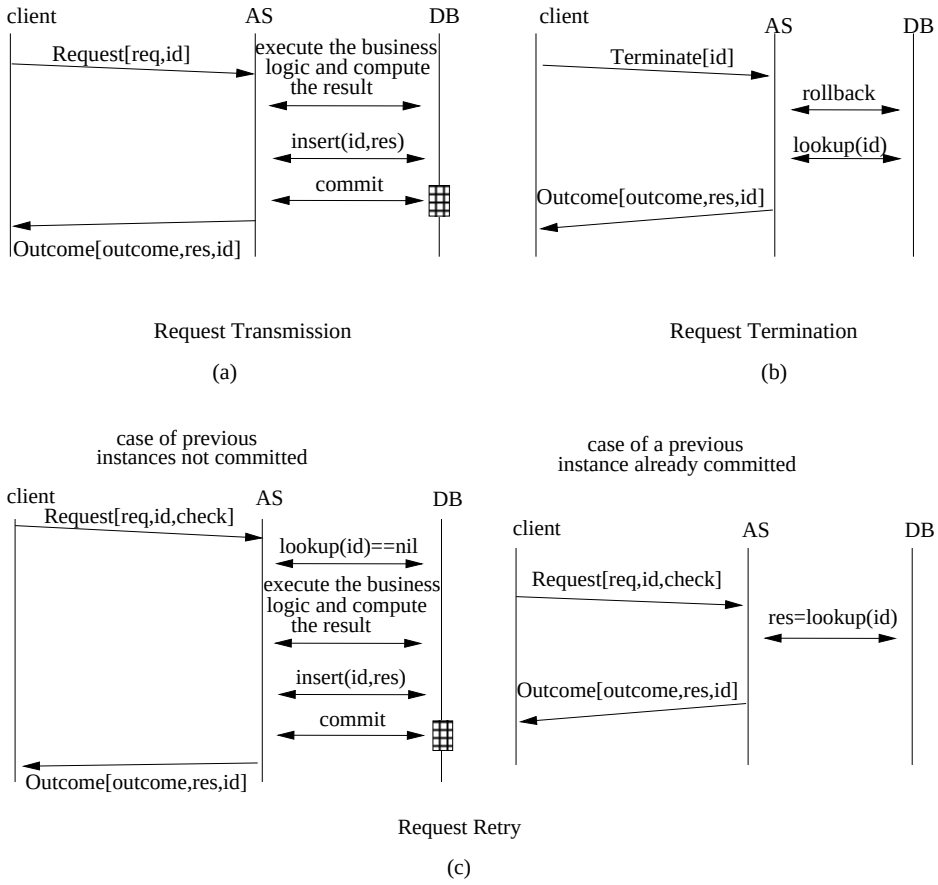


Figure 6.3: Basic Client-Initiated Interactions for the Considered Protocols. (Filled boxes represent eager disk accesses)

request message is always processed before the corresponding terminate messages. This is due to the fact that, when mapping the protocol on conventional technology, according to the specifications of the standard interface for transaction demarcation, namely XA, upon a rollback operation for a transaction with a given identifier, the database system can reuse that identifier for a successive transaction activation (see [117] - state table on page 109). Hence, if a terminate message was processed before the corresponding request message in the FG protocol, the latter message could possibly give rise to a transaction that gets eventually committed. On the other hand, upon the receipt of a reply to a terminate message, the client might activate a new transaction, with a different identifier, which could eventually get committed, thus leading to multiple updates at the database and violating safety. This situation is shown

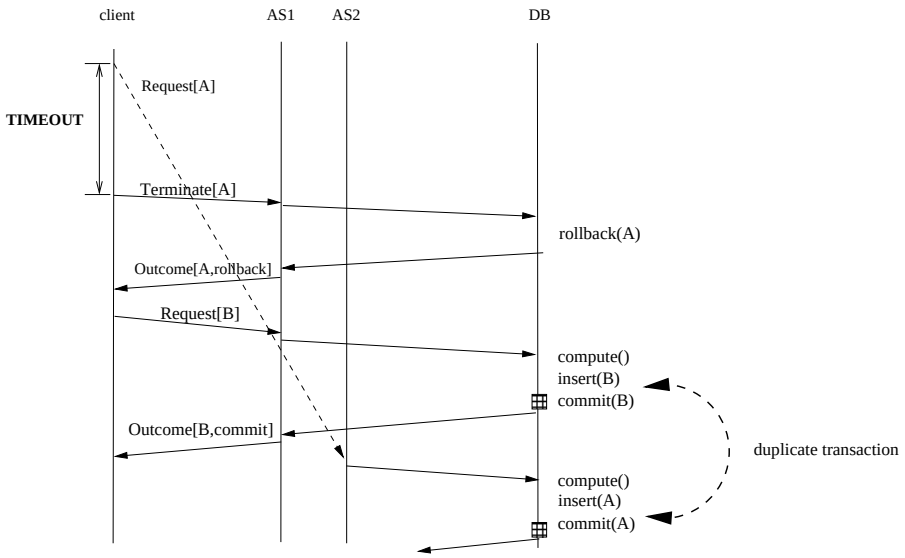


Figure 6.4: Duplicate Transaction Execution in the FG Protocol due to a Violation of the Processing Order of Request and Terminate Messages.

in Figure 6.4, where the long transmission delay for the request message with identifier A causes exactly such an undesirable pattern (i.e. the terminate message for A is processed before the corresponding request message). On the other hand, after the receipt of the reply to the terminate message for A, the client sends a new request message with identifier B, so that both the request messages associated with different identifiers A and B (but containing the same request content) give rise to transactions that are eventually committed. In order to achieve the required processing order constraint for request and terminate messages, the authors suggest to delay the processing of the terminate messages at the application servers. This expedient might reveal adequate in case the application is deployed over an infrastructure with controlled message delivery latency and relative process speed, e.g. a (virtual) private network or an Intranet. However, if the system can experience periods during which the message delivery latency gets unpredictably long and/or the process speeds diverge, e.g. like in an asynchronous system, simply delaying the processing of a terminate message would not suffice to ensure such an ordering constraint. The latter constraint could be enforced through additional mechanisms (e.g. explicit coordination among the servers), but these would negatively affect both performance and scalability of this protocol. By all means, delaying the processing of terminate messages, even if adequate for specific environments, would further penalize the user perceived system responsiveness during the fail-over phase as compared to our solution. Conversely, our protocol does

not require constraints on the processing order of messages exchanged among processes, thus it requires no additional mechanism to enforce such an order and can be straightforwardly adopted in an asynchronous system, e.g. an infrastructure layered on top of public networks over the Internet.

Finally, a drawback deriving from the reliance of the FG protocol on the termination phase is that it needs to assume eventually perfect failure detection to preserve the liveness of the end-to-end interaction. Conversely, our solution does not hinge on any assumption on the accuracy of the failure detection scheme.

6.6.2 Quantitative Comparison with FG

In this section we focus on a quantitative comparison of the user perceived fail-over latencies when employing our solution, extended with the "preventive check optimization" described in Section 6.5, and the FG protocol. For fairness, we avoid considering the parallel invocation version of our protocol (see Section 6.4) in this comparison, given that the structure of the protocol in [43] (i.e., its reliance on a termination phase to handle failure suspicions) prevents its employment in the context of a parallel invocation scheme.

This is done through the introduction of relatively simple analytical models for the protocol behaviors, suitable for comparing them in a wide range of environmental settings. While developing the models we follow a bottom-up approach. Specifically, we first present an analysis of the main client-initiated interactions allowed by the two protocols (e.g. a termination interaction in case of the FG protocol). Latency models for those interactions are used as building blocks for the construction of complete models for the expected end-to-end latency at the client side.

Models for Basic Client-Initiated Interaction

The models we provide in this section express mean latency values for basic client-initiated interactions successfully completed with no timeout expiration at the client side (the effects of timeouts will be included while composing these models to evaluate the whole end-to-end latency perceived by the end-user).

The protocols are based on two different client-initiated interactions. The FG protocol is based on a request transmission interaction, as schematized in Figure 6.3.a, and on a request termination interaction, as schematized in Figure 6.3.b. Note that the request termination interaction can end with either a commit or a rollback indication to the client, depending on whether the transaction was already committed upon the issue of the rollback request by the application server to the database server. Actually, the real outcome is discovered by using the recovery information at the database, which is accessed

via a lookup phase. Our protocol is based on a request transmission interaction, analogous to the one of the FG protocol shown in Figure 6.3.a, and on a request retry interaction, as shown in Figure 6.3.c ([4]). This latter interaction includes the *check* parameter. This allows checking, again through a lookup phase, whether the transaction has been already committed before activating any new instance. In the negative case, the new instance is activated. We denote as P_{commit} the probability that the application server finds the transaction already committed during either the request termination interaction in Figure 6.3.b for the FG protocol or during the request retry interaction in Figure 6.3.c for our protocol. (In other words, P_{commit} indicates the probability that the lookup phase returns with an already established result for the transaction.)

We can now derive expressions for the expected latency of the request transmission interaction in Figure 6.3.a, proper of both protocols, which we denote as T_{req}, and the expected latency of both the request termination and retry interactions (in Figure 6.3.b and in Figure 6.3.b, respectively), each one proper of a specific protocol, which we denote as $T_{terminate}$ and T_{retry}. These expressions are:

$$T_{req} = RTT_{CL/AS} + RTT_{AS/DB} + T_{compute} + T_{insert} \quad (6.1)$$

$$T_{terminate} = RTT_{Cl/AS} + RTT_{AS/DB} + T_{rollback} + T_{lookup} \quad (6.2)$$

$$T_{retry} = RTT_{CL/AS} + RTT_{AS/DB} + T_{lookup} + \\ +(1 - P_{commit})[T_{compute} + T_{insert}] \quad (6.3)$$

where:

- $T_{compute}$ is the average time required to execute the transactional business logic (e.g. SQL statements).
- T_{insert} is the average time required to log the recovery information (i.e. the transaction identifier and the result) at the database.
- $RTT_{CL/AS}$ and $RTT_{AS/DB}$ represent, respectively, the average latency for a request/response interaction between a client and an application server, and between an application server and the database server, respectively.
- $T_{rollback}$ is the time required for handling a forced rollback request for a transaction.

[4]We avoid explicitly modeling the database initiated request retransmission logic (activated by **Task 2** in Figure 6.2), since our study is focused on the evaluation of the perceived latencies during fail-over from the client perspective.

- T_{lookup} represents the time for performing a lookup operation in the table maintaining the recovery information.

In the above expressions, analogously to the analytical model presented in Section 3.5, we have considered the case of transactional logic (including the insertion of the recovery information) executed through a single round trip interaction between application and database servers, e.g., as in stored procedures. This allows avoiding the introduction of an arbitrary delay in the latency models, caused by an arbitrary number of interactions for the management of the transactional logic. Also, with no loss of fairness, we avoid to model transaction aborts due to autonomous decisions of the database server. Note also that, while modeling the FG's request termination interaction, no delay has been introduced for the processing of the terminate request at the application server, which, as discussed in Section 6.6.1, might be required by the FG protocol to ensure correct order of request/terminate message processing. This choice derives from that no clear indication has been provided by the authors on the delay value. However, we underline that omitting this delay even favors the FG protocol in the comparative analysis.

We note that the expression for $T_{terminate}$ in (6.2) does not depend on P_{commit}. This is because the termination phase for the FG protocol has the same pattern independently of whether the transaction the application server attempts to terminate through forced rollback was already committed or not. Anyway, as we shall show in the next section, the parameter P_{commit} plays a role in the expression for the whole end-to-end latency provided by the FG protocol since, in case the termination phase finds the transaction not committed, a new instance of request, with a different identifier, needs to be transmitted by the client.

End-to-end Latency Models

A key factor to build complete end-to-end latency models for the two protocols is the occurrence of a timeout expiration at the client side, i.e. the mechanism employed by the two protocols to support failure detection. The accuracy of such an approach to failure detection is in practice affected by a large number of factors, e.g. the choice of the timeout value with respect to the average system speed, as well as the variance of the average system speed, and the probability of failure of any process involved in the interaction. For simplicity, we will abstract over these details, and model the effects of timeout expiration by a single parameter P_{TO}, namely the probability for a client to experience a timeout during any client-initiated interaction.

As a last preliminary observation, we note that the typical behavior of a Web browser (contacting the Web/application server through HTTP(S)) is to close the underlying TCP connection in case of timeout [116]. Hence,

while deriving end-to-end latency models, we consider the case in which the client-initiated interaction during which timeout is experienced does not get eventually completed (just because the channel for the reply to the client is closed upon timeout expiration).

On the basis of the above considerations, we can express the average user perceived latency for the considered protocols, which we denote as T^{fg} and $T^{our\_protocol}$, as:

$$T^{fg} = (1 - P_{TO})T_{req} + P_{TO}(TO + T^{fg}_{failover}) \quad (6.4)$$

$$T^{our\_protocol} = (1 - P_{TO})T_{req} + P_{TO}(TO + T^{our\_protocol}_{failover}) \quad (6.5)$$

where

- TO is the timeout value at the client side.
- $T^{fg}_{failover}$ and $T^{our\_protocol}_{failover}$ represent the expected latency for the fail-over phase for the two protocols.

By expressions (6.4) and (6.5), the timeout latency TO and the latency for fail-over operations (i.e. $T^{fg}_{failover}$ and $T^{our\_protocol}_{failover}$, respectively) are experienced at the client side only in case of timeout expiration, i.e. with probability P_{TO}. In case of no timeout, the user perceived latency simply consists of the time for a request transmission interaction T_{req} as expressed in (6.1). Also, always for simplicity, but with no loss of fairness and generality, we avoid modeling transaction rollback due to the concurrency control mechanism. This is the reason why, in case of no timeout expiration, the end-to-end latency in both protocols simply corresponds to T_{req}.

To complete the models, we have now to derive expressions for $T^{fg}_{failover}$ and $T^{our\_protocol}_{failover}$. Actually, these expressions can be derived by thinking that fail-over is supported by simply composing client-initiated interactions among those modeled in Section 6.6.2 on a timeout basis. Specifically, the FG protocol lets the client activate request termination interactions on a timeout basis until an outcome is notified to the client. In case the outcome is rollback, the client selects a new request identifier and regenerates its initial behavior by activating a new request transmission interaction. Instead, our protocol lets the client simply activate request retry interactions until one of them is eventually completed with positive outcome for the transaction. As a consequence, the expected latency for the fail-over can be expressed for the two protocols as follows:

$$T^{fg}_{failover} = (1 - P_{TO})[T_{terminate} + (1 - P_{commit})T^{fg})] + \\ + P_{TO}(TO + T^{fg}_{failover}) \quad (6.6)$$

$$T^{our\_protocol}_{failover} = (1 - P_{TO})T_{retry} + P_{TO}(TO + T^{our\_protocol}_{failover}) \quad (6.7)$$

$T_{compute} + T_{insert}$	T_{lookup}	$T_{rollback}$
195,18	1.42	0.55

Figure 6.5: Measured Parameters Values (Expressed in ms).

By means of simple algebraic transformations and replacements, we finally obtain the following expressions for the end-to-end latency provided by the two protocols:

$$T^{fg} = \frac{(1-P_{TO})T_{req} + P_{TO}(TO + T_{terminate} + \frac{P_{TO}}{1-P_{TO}}TO)}{1-(1-P_{commit})P_{TO}} \quad (6.8)$$

$$T^{our\_protocol} = (1-P_{TO})T_{req} + P_{TO}(TO + T_{retry} + \frac{P_{TO}TO}{1-P_{TO}}) \quad (6.9)$$

Parameter Treatment

Before presenting and discussing quantitative results deriving from the developed models in (6.8) and (6.9), we provide indications on how the parameters appearing in the models have been treated in our analysis.

In order to use realistic values for $T_{compute}$, T_{insert}, $T_{rollback}$ and T_{lookup}, we have developed prototype implementations of (i) basic modules supporting the actions required by the protocols for the manipulation of the recovery information, and of (ii) the TPC BENCHMARK^{TM} C (New-Order-Transaction), already used in the performance evaluation study presented in Chapter 3. The table in Figure 6.5 lists the costs of the activity on the back-end database, which have been measured using the same software/hardware configuration already reported in Section 3.5. Also in this case, each reported value, expressed in milliseconds, is the average over a number of samples that ensures confidence interval of 10% around the mean at the 95% confidence level.

For what concerns P_{TO}, we selected the realistic value of 0.01, reflecting the fact that, in practical settings, only a limited percentage of interactions experience timeout. P_{commit} was set to the value of 0.5, indicating that, during fail-over, we equally likely find the transaction already committed or uncommitted.

For what concerns the parameters $RTT_{CL/AS}$ and $RTT_{AS/DB}$, we note that they are typically dependent on the relative locations of clients, application servers and database server. In the analysis we consider the following three classical scenarios for what concerns the deployment of the different system components in a multi-tier system (see Section 1.2.2):

Scenario-A: Clients, application and database servers are all geographically distributed and communicate with each other through the Internet.

	$RTT_{c/as}$	$RTT_{as/db}$
Scenario-A	150	150
Scenario-B	150	5
Scenario-C	5	5

Figure 6.6: Communication Latency Values (Expressed in ms).

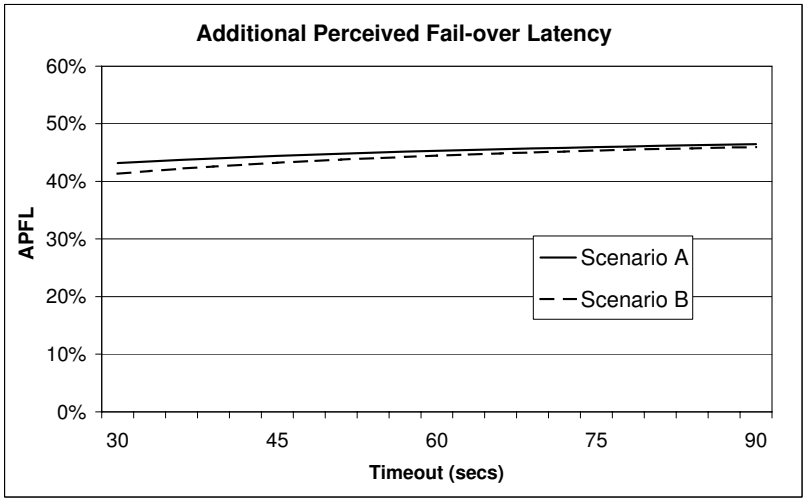
Scenario-B: Geographically spread clients, connected to the application servers through the Internet. Application and database servers residing on the same local area network.

Scenario-C: Clients, application and database servers residing on the same local area network, as in the case of Intranet applications.

Results

The table in Figure 6.5 shows the considered values for $RTT_{CL/AS}$ and $RTT_{AS/DB}$, which have been chosen as representative for the corresponding scenarios [59]. (We recall that, as discussed in Section 6.6.1, the FG protocol needs assumptions on the message processing order that may not hold for some of the considered scenarios. We include these scenarios in the comparison for the sake of completeness. Such a choice also allows us to show that, even if those assumptions were guaranteed without any additional penalty for the FG protocol, e.g. without explicitly coordinating process activities, which is in fact not modeled by expression (6.8), our protocol would anyway outperform that solution.) For each scenario we leave the value of TO as the independent parameter of the performance study. This allows us to compare the two protocols when considering settings with different features for the variance of the end-to-end interaction latency. Specifically, lower values for TO are representative of settings with highly predictable performance (e.g., a closed system with a limited number of clients), for which suspects of failures can be reasonably triggered on the basis of relatively aggressive timeouts. On the other hand, longer timeout values are representative of situations with much larger fluctuations (and hence variance) for the end-to-end response time (as in systems layered on top of best effort public infrastructures), for which the timeout values need to be more conservative in order not to incur excessive false failure suspicions. To reflect the different degrees of variance in the expected latencies characterizing the three considered scenarios, we let TO vary in the range [30-90] for Scenario-A and Scenario-B (where clients interact with the system across the Internet), and in the range [10-30] for Scenario-C (where clients are on the same local area network also hosting the server side components).

The purpose of our analysis is to provide quantitative indications on the impact of the two considered protocols on the user perceived latency during



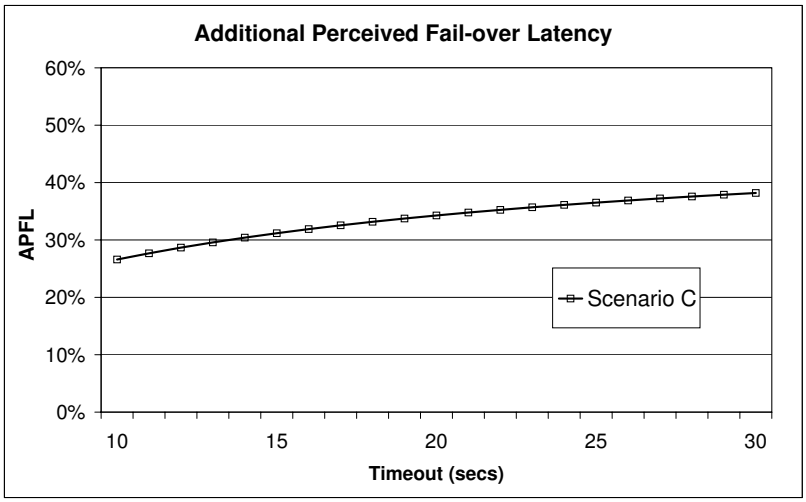


Figure 6.7: Additional Perceived Fail-Over Latency.

	Baseline	Our protocol	Overhead
TPC-C (New Order Transaction)	192.84	195.18	+1.21%

Table 6.1: Quantification of the Inherent Cost of Reliability (Values Expressed in ms).

fail-over. To this aim, we use the end-to-end latency models previously presented to plot the Additional Perceived Fail-Over Latency (APFL) of the FG protocol compared to our proposal, defined as:

$$\text{APFL} = \frac{T_{failover}^{fg} - T_{failover}^{our\_protocol}}{T_{failover}^{our\_protocol}} \tag{6.10}$$

In Figure 6.7 we report the plots of the APFL for the three considered scenarios. Looking at the results, we note that our protocol provides fail-over latency which is between the 25% and the 50% lower than the one of the protocol in [43], with the largest advantages achieved in Scenario-A and Scenario-B. This depends on that the Internet latency between client and application servers have a penalizing impact on the additional termination phase required by the FG in case of timeout expiration.

It is also interesting to note that, for all the considered scenarios, the APFL curve is almost flat while varying the selected timeout values. This points out how our proposal can be adopted to provide more responsive fail-over for the case of both aggressive timeout values (suited for the case of predictable system performance, i.e., low response time variance) and conservative timeout values (suited for systems with higher response time variance).

6.6.3 Comparison with the Baseline Protocol

In this section we provide indications on the inherent cost of reliability when using our protocol, namely the overhead imposed by our proposal with respect to a baseline protocol which does not ensure any reliability guarantee. Such a baseline protocol simply processes the client request within a transaction to be performed by the middle-tier. In other words, the two protocols differ only because in our solution an additional tuple is inserted in a user level database table within the same transaction being executed on behalf of the application business logic.

We focus on the overhead imposed by our protocol in terms of additional processing time required by the database server to handle the insertion of the recovery information within the database table. In other words, the overhead evaluation we provide does not consider the communication latency among processes. We note, however, that the real overhead would be even lower than

the values we report if evaluated over the whole end-to-end latency instead of simply the time to process the transaction at the database server.

We report in Table 6.1 the average processing time for SQL statements associated with the New-Order-Profile of the TPC BENCHMARK^{TM} C, in a separate row, the exact cost for inserting the recovery information in a database table. All these measures were obtained by means of the previously described test system. These data clearly show that the overhead exhibited by our protocol is minimal.

Bibliography

[1] M. Allman. A web server's view of the transport layer. *ACM Computer Communication Review*, 30(5):10–20, 2000.

[2] C. Almeida and P. Veríssimo. Using light-weight groups to handle timing failures in quasi-synchronous systems. In *Proc. of the 19th Real-Time Systems Symposium (RTSS)*, pages 430–439, 1998.

[3] C. Amza, A. L. Cox, and W. Zwaenepoel. Data replication strategies for fault tolerance and availability on commodity clusters. In *Proc. of the Conference on Dependable Systems and Networks (DSN)*, pages 459–472, 2000.

[4] D. Andersen, H. Balakrishnan, M. Kaashoek, and R. Morris. The case for resilient overlay networks. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems*, pages 152 – 157, 2001.

[5] M. Andreolini, M. Colajanni, and R. Morselli. Performance study of dispatching algorithms in multi-tier web architectures. *SIGMETRICS Performance Evaluation Review*, 30(2):10–20, 2002.

[6] T. Anker, D. Dolev, G. Greenman, and I. Shnayderman. Evaluating total order algorithms in WAN. In *Proc. of the International Workshop on Large-Scale Group Communication at SRDS 2003*, 2003.

[7] J. Apostolopoulos and M. Trott. Path diversity for enhanced media streaming. *IEEE Communications Magazine*, 42(8):80 – 87, 2004.

[8] J. Apostolopoulos, T. Wong, W. Tan, and S. Wee. On multiple description streaming with content delivery networks. In *Proc. of the 21th Conference on Computer Communications (INFOCOM)*, volume 3, pages 1736 – 1745. IEEE Computer Society Press, 2002.

[9] A. Banerjea. Simulation study of the capacity effects of dispersity routing for fault tolerant realtime channels. In *Proc. of the 2nd Conference Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, pages 194–205. ACM Press, 1996.

[10] P. Barford, A. Bestavros, A. Bradley, and M. Crovella. Changes in web client access patterns - characteristics and caching implications. In *Proc. of the 8th World Wide Web Conference (WWW)*, volume 2, pages 15–18, 1999.

[11] R. Barga, S. Chen, and D. B. Lomet. Improving logging and recovery performance in Phoenix/App. In *Proc. of the 20th International Conference on Data Engineering (ICDE)*, pages 486–497. IEEE Computer Society Press, 2004.

[12] R. Barga, D. Lomet, S. Agrawal, and T. Baby. Persistent client-server database sessions. In *Proc. of the 7th Conference on Extending Database Technology (EDBT)*, pages 462–477. Springer, 2000.

[13] R. Barga, D. Lomet, G. Shegalov, and G. Weikum. Recovery guarantees for internet applications. *ACM Transactions on Internet Technology*, 4(3):289–328, 2004.

[14] P. A. Bernstein and N. Goodman. Multiversion concurrency control: theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[15] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc., 1987.

[16] P. A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *Proc. of the 19th Conference on the Management of Data (SIGMOD)*, pages 101–112. ACM, May 1990.

[17] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing: for the Systems Professional.* Morgan Kaufmann Publishers Inc., 1997.

[18] A. Bestavros. An adaptive information dispersal algorithm for time critical reliable communication. *Network Management and Control*, 2:423–438, 1994.

[19] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. In *Proc. of the 9th World Wide Web Conference (WWW)*, 1–16, 2000.

[20] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[21] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. *The primary-backup approaches.* ACM Press/Addison-Wesley Publishing Co., 1993.

[22] J. W. Byers, M. Luby, and M. Mitzenamcher. Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads. In *Proc. of the 18th Conference on Computer Communications (INFOCOM)*, pages 275–283. IEEE Computer Society Press, 1999.

[23] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The state of the art in locally distributed Web-server systems. *ACM Computer Surveys*, 34(2):263–311, 2002.

[24] N. Cardwell, S. Savage, and T. Anderson. Modeling the performance of short TCP connections. Technical Report November, Computer Science Department, Washington University, 1998.

[25] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Communications of the ACM*, 43(2):225–267, 1996.

[26] K. M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

[27] J. Chen. *New Approaches to Routing for Large-Scale Data Networks*. PhD thesis, Rice University, 1999.

[28] L. Cherkasova. Optimizing the reliable distribution of large files within CDNs. In *Proc. of the 10th International Symposium on Computers and Communications (ISCC)*, pages 692–697. IEEE Computer Society Press, 2005.

[29] R. Christodoulopoulou, K. Manassiev, A. Bilas, and C. Amza. Fast and transparent recovery for continuous availability of cluster-based servers. In *Proc. of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 221–229. ACM Press, 2006.

[30] R. Collins and J. Plank. Downloading replicated, wide-area files - a framework and empirical evaluation. In *Proc. of the 3th Symposium on Network Computing and Applications (NCA)*, pages 89–96. IEEE Computer Society Press, 2004.

[31] G. F. Coulouris and J. Dollimore. *Distributed systems: concepts and design*. Addison-Wesley Longman Publishing Co., Inc., 1988.

[32] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[33] A. Datta, K. Dutta, H. M. Thomas, D. E. VanderMeer, K. Ramamritham, and D. Fishman. A comparative study of alternative middle

tier caching solutions to support dynamic web content acceleration. In *Proc. of the 27th Conference on Very Large Data Bases (VLDB)*, pages 667–670. Morgan Kaufmann, 2001.

[34] S. B. Davidson, V. Wolfe, and I. Lee. Timed atomic commitment. *IEEE Transactions on Computers*, 40(5):573–583, 1991.

[35] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[36] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[37] D. Dolev and H. Strong. Requirements for agreement in a distributed system. In H. Schneider, editor, *Distributed Data Bases*, pages 115–129. Amsterdam: North-Holland, 1982.

[38] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[39] P. A. Felber, B. Garbinato, and R. Guerraoui. The design of a CORBA group communication service. In *Proc. of the 15th Symposium on Reliable Distributed Systems (SRDS)*, pages 150–161, IEEE Computer Society Press 1996.

[40] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Proc. of the 4th Conference on Fundamentals of Computation Theory (FCT)*, pages 127–140. Springer-Verlag, 1983.

[41] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[42] J. Freytag, F. Cristian, and B. Kähler. Masking system crashes in database application programs. In *Proc. of the 13th Conference on Very Large Data Bases (VLDB)*, pages 407–416, 1987.

[43] S. Frølund and R. Guerraoui. A pragmatic implementation of e-Transactions. In *Proc. of the 19th Symposium on Reliable Distributed Systems (SRDS)*, pages 186–195. IEEE Computer Society Press, 2000.

[44] S. Frølund and R. Guerraoui. Implementing e-Transactions with asynchronous replication. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):133–146, feb 2001.

[45] S. Frølund and R. Guerraoui. e-Transactions: End-to-end reliability for three-tier architectures. *IEEE Transaction on Software Engineering*, 28(4):378–395, 2002.

[46] G. Gama, K. Nagaraja, R. Bianchini, R. Martin, W. Meira Jr., and T. Nguyen. State maintenance and its impact on the performability of multi-tiered internet services. In *Proc. of the 23rd Symposium on Reliable Distributed Systems (SRDS)*, pages 146–158, 2004.

[47] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of the 16th Conference on Management of Data (SIGMOD)*, pages 249–259. ACM Press, 1987.

[48] F. C. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, 1999.

[49] J. Gray. The transaction concept: Virtues and limitations (invited paper). In *Proc. of the 7th Conference on Very Large Data Bases (VLDVB)*, pages 144–154. IEEE Computer Society Press, 1981.

[50] J. Gray. Why do computers stop and what can we do about it? In *Proc. of 6th Conference on Reliability and Distributed Databases*, pages 3–12. IEEE Computer Society Press, 1987.

[51] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the 25th Conference on the Management of Data (SIGMOD)*, pages 173–182. ACM, 1996.

[52] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1991.

[53] R. Guerraoui, M. Hurfin, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper. Consensus in asynchronous distributed systems: A concise guided tour. In S. S. S. Krakowiak, editor, *Advances in Distributed Systems*, volume LNCS 1752, pages 33–47. Spinger Verlang, 2000.

[54] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. Wetherall. Improving the reliability of internet paths with one-hop source routing. In *Proc. of the 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 183–198. Usenix, 2004.

[55] R. Gupta, J. R. Haritsa, and K. Ramamritham. More optimism about real-time distributed commit processing. In *Proc. of the 18th Real-Time Systems Symposium (RTSS)*, pages 123–133. IEEE Computer Society, 1997.

[56] R. K. Gupta. Commit processing in distributed on-line and real-time transaction processing systems. Master's thesis, Supercomputer Education and Research Centre, Indian Institute of Science, 1997.

[57] J. Haritsa and K. Ramamritham. Adding PEP to real-time distributed commit processing. In *Proc. of the 21th Real-Time Systems Symposium (RTSS)*, pages 37–46. IEEE Computer Society, 2000.

[58] J. R. Haritsa, K. Ramamritham, and R. Gupta. The PROMPT real-time commit protocol. *IEEE Transactions on Parallel and Distributed Systems*, 11(2):160–181, 2000.

[59] Internet Traffic Report. http://www.internettrafficreport.com.

[60] R. Jimenez-Peris, M. Patino-Martinez, and S. Arivalo. Deterministic scheduling for transactional multithreaded replicas. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 164–173. IEEE Computer Society Press, 2000.

[61] K. Johnson, J. Carr, M. Day, and M. Kaashoek. The measured performance of content distribution networks. In *Proc. of the 5th Workshop on Web Caching and Content Delivery*, 2000.

[62] M. Kalyanakrishnan, R. K. Iyer, and J. U. Patel. Reliability of internet hosts: a case study from the end user's perspective. *Computing Networks*, 31(1-2):47–57, 1999.

[63] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults. *SIGACT News*, 32(2):45–63, 2001.

[64] S. Kim, S. H. Son, and J. A. Stankovic. Performance evaluation on a real-time database. In *Proc. of the 8th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 253–265. IEEE Computer Society Press, 2002.

[65] K.-W. Lam, S. H. Son, S.-L. Hung, and Z. Wang. Scheduling transactions with stringent real-time constraints. *Information Systems*, 25(6):431–452, 2000.

[66] L. Lamport and N. A. Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 1157–1199. Elsevier and MIT Press, 1990.

[67] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[68] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. M. Dias. Design and performance of a web server accelerator. In *Proc. of the 18th Conference on Computer Communications (INFOCOM)*, pages 135–143, 1999.

[69] W.-S. Li, W.-P. Hsiung, D. V. Kalashnikov, R. Sion, O. Po, D. Agrawal, and K. S. Candan. Issues and evaluations of caching solutions for web application acceleration. In *Proc. of the 28th Conference on Very Large Data Bases (VLDB)*, pages 1019–1030. Morgan Kaufmann, 2002.

[70] M. Little and S. Shrivastava. Integrating the object transaction service with the Web. In *Proc. of the 2nd Int. Workshop on Enterprise Distributed Object Computing*, pages 194–205. IEEE Computer Society Press, 1998.

[71] F. Liu, Y. Zhao, W. Wang, and D. Makaroff. Database server workload characterization in an e-commerce environment. In *Proc. of the 12th Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)*, pages 475–483, 2004.

[72] D. Llambiri, A. Totok, and V. Karamcheti. Efficiently distributing component-based applications across wide-area environments. In *Proc. of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, pages 412–421. IEEE Computer Society Press, 2003.

[73] D. B. Lomet. Persistent middle tier components without logging. In *Proc. of the 9th International Database Engineering and Applications Symposium (IDEAS)*, pages 37–46. IEEE Computer Society Press, 2005.

[74] D. B. Lomet and G. Weikum. Efficient and transparent application recovery in client-server information systems. In *Proc. of the 27th Conference on Management of Data (SIGMOD)*, pages 460–471. ACM Press, 1998.

[75] D. Long, J. L. Carroll, and C. J. Park. A study of the reliablility of internet sites. In *Proc. of the 10th Symposium on Reliable Distributed Systems (SRDS)*, pages 177–186, 1991.

[76] D. Long, A. Muir, and R. Golding. A longitudinal survey of internet host reliability. In *Proc. of the 14th Symposium on Reliable Distributed Systems (SRDS)*, pages 2–9. IEEE Computer Society Press, 1995.

[77] M. Y. Luo and C. S. Yang. Constructing zero-loss web services. In *Proc. of the 20th Conference on Computer Communications (INFOCOM)*, pages 1781–1790, 2001.

[78] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[79] S. Maffeis. Adding group communication and fault-tolerance to CORBA. In *Proc. of the 2nd USENIX Conference on Object-Oriented Technologies*, 1995.

[80] N. F. Maxemchuck. *Dispersity Routing in Store and Forward Networks*. PhD thesis, University of Pennsylvania, 1975.

[81] K. Mazouni, B. Garbinato, and R. Guerraoui. Filtering duplicated invocations using symmetric proxies. In *Proc. of the 4th International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, pages 118–126. IEEE Computer Society Press, 1995.

[82] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: An approach to universal topology generation. In *Proc. of the 9th Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)*, pages 346 – 353. IEEE Computer Society Press, 2001.

[83] N. C. Mendona and J. A. F. Silva. An empirical evaluation of clientside server selection policies for accessing replicated web services. In *Proc. of the 20th ACM Symposium on Applied Computing (SAC)*, pages 718–723. ACM Press, 2005.

[84] C. Mohan. Caching technologies for web applications. In *Proc. of the 27th Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, 2001.

[85] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Transactions on Database Sstems*, 11(4):378–396, 1986.

[86] P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, University of California, Santa Barbara, USA, 1999.

[87] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Exploiting the Internet Inter-ORB Protocol interface to provide CORBA with fault tolerance. In *Proc. of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 81–90. USENIX, 1997.

[88] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded corba applications. In *Proc. of the 18th Symposium on Reliable Distributed Systems (SRDS)*, pages 263–273. IEEE Computer Society Press, 1999.

[89] NLANR. National laboratory for applied network research. http://www.nlanr.net/Routing/rawdata.

[90] OMG Document Number 95-3-31. CORBAservices: Common Object Services Specification, 1995.

[91] S. Pakin, M. Lauria, and A. Chen. High performance messaging on workstations: Illinois fast message (FM) for myrinet. In *Proc. of the 9th International Conference on Supercomputing (ICS)*. ACM Press, 1995.

[92] C. Park, S. Park, and S. H. Son. Multiversion locking protocol with freezing for secure real-time database systems. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1141–1154, 2002.

[93] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proc. of the 14th Conference on Distributed Computing (DISC)*, Lecture Notes in Computer Science, pages 315–329. Springer, 2000.

[94] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications.,. In H.-A. Jacobsen, editor, *Proc. of the 4th ACM/IFIP/USENIX Middleware Conference*, volume 3231 of *Lecture Notes in Computer Science*, pages 155–174. Springer, 2004.

[95] S. Pleisch, A. Kupšys, and A. Schiper. Preventing orphan requests in the context of replicated invocation. In *Proc. of the 22nd the Symposium on Reliable Distributed Systems (SRDS)*, pages 119 – 128. IEEE Computer Society Press, 2003.

[96] D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice Hall PTR, 2003.

[97] M. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[98] M. Raynal and M. Singhal. Mastering agreement problems in distributed systems. *IEEE Software*, 18(4):40–47, 2001.

[99] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1206–1217, 2003.

[100] P. Romano, F. Quaglia, and B. Ciciani. A protocol for improved user perceived QoS in web transactional applications. In *Proc. of the 3th Symposium on Network Computing and Applications (NCA)*, pages 69–76. IEEE Computer Society Press, 2004.

[101] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[102] A. Schiper and A. Ricciardi. Virtually-synchronous communication based on a weak failure suspector. In *Proc. of the 27th Symposium on Fault-Tolerant Computing (FTCS)*, pages 534–543, 1993.

[103] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computuer Surveys*, 22(4):299–319, 1990.

[104] F. B. Schneider. *Replication management using the state-machine approach*, chapter 7, pages 169–197. ACM Press/Addison-Wesley Publishing Co., 1993.

[105] B. A. Shirazi, K. M. Kavi, and A. R. Hurson. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[106] L. C. Shu, J. A. Stankovic, and S. H. Son. Achieving bounded and predictable recovery using real-time logging. In *Proc. of the 8th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 286–297. IEEE Computer Society Press, 2002.

[107] M. Singhal. Update transport: A new technique for update synchronization in replicated database systems. *IEEE Transactions on Software Engineering*, 16(12):1325–1336, 1990.

[108] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-based content routing using xml. In *Proc. of the 18th Symposium on Operating Systems Principles (SOSP)*, pages 160–173. ACM Press, 2001.

[109] S. H. Son and K.-D. Kang. Qos management in web-based real-time data services. In *Proc. of the 4th Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS)*, pages 129–136. IEEE Computer Society Press, 2002.

[110] J. Stamos and F. Cristian. Coordinator log transaction execution protocol. *Distributed and Parallel Databases*, 1(4):383–408, 1993.

[111] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, 5(3):188–194, 1979.

[112] S. Subramanian and M. Singhal. A real-time protocol for stock market transactions. In *Proc. of the 1st Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS)*, pages 2–10. IEEE Computer Society Press, 1999.

[113] Sun Microsystems. Sun Java System Message Queue v3.5 SP1 Administration Guide.

[114] Sun Microsystems. JSR 220: Enterprise JavaBeans^{TM}, version 3.0 - Java Persistence API, May 2006.

[115] Suresha and J. R. Haritsa. On reducing dynamic web page construction times. In *Proc. of the 6th Asia-Pacific Web Conference (APWeb)*, Lecture Notes in Computer Science, pages 722–731. Springer, 2004.

[116] The Jakarta Project. Jakarta commons: HTTP client.

[117] The Open Group. *Distributed TP: The XA+ Specification Version 2.* 1994.

[118] The PostgreSQL Global Development Group. Multiversion concurrency control. In *PostgreSQL 7.2.1 Documentation*, 2001.

[119] Transaction Processing Performance Council. *TPC Benchmark^{TM} C, Standard Specification, Revision 5.1.* Transaction Processing Perfomance Council, 2002.

[120] Transaction Processing Performance Council. *TPC Benchmark^{TM} W, Standard Specification, Version 1.8.* Transaction Processing Perfomance Council, 2002.

[121] A. Vakali and G. Pallis. Content delivery networks: Status and trends. *IEEE Internet Computing*, 07(6):68–74, 2003.

[122] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J. Barrera, and J. Gray. The design and architecture of the Microsoft cluster service - a practical approach to high availability and scalability. In *Proc. of the 28th Internation Symposium on Fault-Tolerant Computing Systems (FTCS)*, pages 422–431, 1998.

[123] Y. Wei, S. H. Son, and J. A. Stankovic. Maintaining data freshness in distributed real-time databases. In *Proc. of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 251–260. IEEE Computer Society Press, 2004.

[124] Y. Wei, S. H. Son, J. A. Stankovic, and K. D. Kang. Qos management in replicated real time databases. In *Proc. of the 24th Real-Time Systems Symposium (RTSS)*, pages 86–97. IEEE Computer Society Press, 2003.

[125] J. Wenyu and H. Schulzrinne. Modeling of packet loss and delay and their effect on real-time multimedia service quality. In *Proc. of the 10th Network and Operating System Support for Digital Audio and Video (NOSSDAV)*. ACM Press, 2000.

[126] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative web proxy caching. In *Proc. of the 6th Symposium on Operating Systems Principles (SOSP)*. ACM Press, 1999.

[127] H. Wu and B. Kemme. Fault-tolerance for stateful application servers in the presence of advanced transactions patterns. In *Proc. of the 24th Symposium on Reliable Distributed Systems (SRDS)*, pages 95 – 108. IEEE Computer Society Press, 2005.

[128] H. Wu, B. Kemme, and V. Maverick. Eager replication for stateful J2EE servers. In *Proc. of the 6th CoopIS, DOA, and ODBASE, OTM Confederated Conferences*, pages 1376–1394. LNCS 3291, Springer Verlang, 2004.

[129] M. Xiong, K. Ramamritham, J. Haritsa, and J. Stankovic. MIRROR: A state-conscious concurrency control protocol for replicated real-time databases. *Information Systems*, 27(4):27–297, 2002.

[130] Zona Research. The need for speed. Technical report, Zona Research, 1999.