Universidade do Minho
Escola de Engenharia
Departamento de Informática

Partial Replication in the Database State Machine

por

António Luís Pinto Ferreira de Sousa

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

Orientador:
Rui Carlos Mendes de Oliveira
(Professor Associado, Universidade do Minho)

Braga
Setembro de 2006

Este trabalho foi parcialmente financiado pela Fundação para a Ciência e a Tecnologia através do projecto ESCADA (POSI/CHS/33792/2000).

Abstract

Enterprise information systems are nowadays commonly structured as multi-tier architectures and invariably built on top of database management systems responsible for the storage and provision of the entire business data. Database management systems therefore play a vital role in today's organizations, from their reliability and availability directly depends the overall system dependability.

Replication is a well known technique to improve dependability. By maintaining consistent replicas of a database one can increase its fault tolerance and simultaneously improve system's performance by splitting the workload among the replicas.

In this thesis we address these issues by exploiting the partial replication of databases. We target large scale systems where replicas are distributed across wide area networks aiming at both fault tolerance and fast local access to data. In particular, we envision information systems of multinational organizations presenting strong access locality in which fully replicated data should be kept to a minimum and a judicious placement of replicas should be able to allow the full recovery of any site in case of failure.

Our research departs from work on database replication algorithms based on group communication protocols, in detail, multi-master certification-based protocols. At the core of these protocols resides a total order multicast primitive responsible for establishing a total order of transaction execution.

A well known performance optimization in local area networks exploits the fact that often the definitive total order of messages closely following the spontaneous network order, thus making it possible to optimistically proceed in parallel with the ordering protocol. Unfortunately, this optimization is invalidated in wide area networks, precisely when the increased latency would make it more useful. To overcome this we present a novel total order protocol with optimistic delivery for wide area networks. Our protocol uses local statistic estimates to independently order messages closely matching the definitive one thus allowing optimistic execution in real wide area networks.

Handling partial replication within a certification based protocol is also particularly challenging as it directly impacts the certification procedure itself. Depending on the approach, the added complexity may actually defeat the purpose of partial replication. We devise, implement and evaluate two variations of the Database State Machine protocol discussing their benefits and adequacy with the workload of the standard TPC-C benchmark.

Resumo

Os sistemas de informação empresariais actuais estruturam-se normalmente em arquitecturas de software multi-nível, e apoiam-se invariavelmente sobre um sistema de gestão de bases de dados para o armazenamento e aprovisionamento de todos os dados do negócio. A base de dado desempenha assim um papel vital, sendo a confiabilidade do sistema directamente dependente da sua fiabilidade e disponibilidade.

A replicação é uma das formas de melhorar a confiabilidade. Garantindo a coerência de um conjunto de réplicas da base de dados, é possível aumentar simultaneamente a sua tolerância a faltas e o seu desempenho, ao distribuir as tarefas a realizar pelas várias réplicas não sobrecarregando apenas uma delas.

Nesta tese, propomos soluções para estes problemas utilizando a replicação parcial das bases de dados. Nos sistemas considerados, as réplicas encontram-se distribuídas numa rede de larga escala, almejando-se simultaneamente obter tolerância a faltas e garantir um acesso local rápido aos dados. Os sistemas propostos têm como objectivo adequarem-se às exigências dos sistemas de informação de multinacionais em que em cada réplica existe uma elevada localidade dos dados acedidos. Nestes sistemas, os dados replicados em todas as réplicas devem ser apenas os absolutamente indispensáveis, e a selecção criteriosa dos dados a colocar em cada réplica, deve permitir em caso de falha a reconstrução completa da base de dados.

Esta investigação tem como ponto de partida os protocolos de replicação de bases de dados utilizando comunicação em grupo, em particular os baseados em certificação e execução optimista por parte de qualquer uma das réplicas. O mecanismo fundamental deste tipo de protocolos de replicação é a primitiva de difusão com garantia de ordem total, utilizada para definir a ordem de execução das transacções.

Uma optimização normalmente utilizada pelos protocolos de ordenação total é a utilização da ordenação espontânea da rede como indicador da ordem das mensagens, e usar esta ordem espontânea para processar de forma optimista as mensagens em paralelo com a sua ordenação. Infelizmente, em redes de larga escala a espontaneidade de rede é praticamente residual, inviabilizando a utilização desta optimização precisamente no cenário em que a sua utilização seria mais vantajosa. Para contrariar esta adversidade propomos um novo protocolo de ordenação total com entrega optimista para redes de larga escala. Este protocolo utiliza informação estatística local a cada processo para "produzir" uma ordem espontânea muito mais coincidente com a ordem total obtida viabilizando a utilização deste tipo de optimizações em redes de larga escala.

Permitir que protocolos de replicação de bases de dados baseados em certificação suportem replicação parcial coloca vários desafios que afectam directamente a forma com é executado o procedimento de certificação. Dependendo da abordagem à replicação parcial, a complexidade gerada pode até comprometer os propósitos da replicação parcial. Esta tese concebe, implementa e avalia duas variantes do protocolo da *database state machine* com suporte para replicação parcial, analisando os benefícios e adequação da replicação parcial ao teste padronizado de desempenho de bases de dados, o TPC-C.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In today's globalized world, it is common to organizations be present in several countries or continents. It is also increasing the reliance of organizations on theirs information system whose dependability becomes critical with the growth and internationalization of the organization.

In a globalization scenario, it is required, due to the presence of the organization in different zones of the globe, that its information system must be always on-line. The unavailability of the information system even for a few minutes may represent a significant loss in the organization's business, and in an extreme scenario may also imply its collapse.

To fulfill the requirement of being always on-line the information system must be fault tolerant, i.e. it must survive the failure of any of its components. In a globalization scenario, tolerating the failure of any of its components may not be sufficient, i.e., hosting the information system in a dependable cluster may not be sufficient. The information system must survive events such as earth-quakes, tsunamis or any other catastrophic event either natural or not. This requires the components of the information system to be spread over a wide area network, ensuring that the disappearing of a location is not sufficient to break the information system.

In the development of the organizations' information systems it is common to use multi-tier architectures, typically 3-tier architectures, usually referred as the *Presentation* tier, the *Logical* or *Application* tier, and the *Data* tier. The Data tier is the heart of organizations' information systems as it is the responsible to store and make available all the organization's data.

Replication is a well known technique to increase the dependability of a system component, either hardware or software. Regarding the Data tier of the referred architecture, it usually uses a database to store the organization's data. In order to achieve the desired degree of dependability, replicated databases are the candidates to be used in the Data tier. The use of replicated databases in such an environment, i.e., in a wide area network poses several challenges.

Currently replicated databases are usually targeted to clusters or local area networks and most of the available products relax consistency in favor of better performance, i.e., most of the replicated databases do not ensure strong data consistency among the replicas.

In a globalized scenario, an organization to achieve the desired degree of dependability, must have replicas of their most sensitive data in several locations of a wide area network. This new kind of environment, with high point to point latency and limited bandwidth, poses several challenges to replicated databases that have been targeted to local environments usually with low latency and without bandwidth restrictions.

Another issue relating to database replication that deserve some attention is the data being replicated. In a globalized organization, some of its data is relevant for all of its branches, but surely there is other data that is only relevant to one or some of the branches but not to all of the branches. Given that some data is not relevant to some of the branches of the organization, it is questionable why to replicated all data to every replica. Should only relevant data be replicated to each replica, i.e., using partial replication instead of full replication, and it would be possible to achieve a high degree of dependability, and reduce the required network bandwidth and also local storage at each replica.

Given the lack of database replication proposals for wide area networks, ensuring strong consistency and supporting partial replication, the goal of this thesis is to address and propose solutions for the problems arising from partially replicated databases and wide area networks.

1.1 Group Based Database Replication

Replication as long been a research issue either in the distributed systems community [PCD91, Sch93, BMST93, DSS98] either in the databases community [Tho79, Gif79, TGGL82, GSC$^+$83, ES83, GMB85, Pâr86, Her87, vRT88, Pâr89, BGMS89, JM90, PL91, Kum91, AAC91, CAA92, WB92, KRS93, TPK95, PW97, NW98]. Although pursuing the same objective of ensuring replicas consistency and augment the system dependability, the protocols developed by the distributed systems and databases communities concentrate in a particular aspect of the problem, which results in very different and unrelated protocols.

Replication protocols in the database community concentrate mainly on the data and its semantic. The properties of the communication primitives and how they can help in ensuring data consistency are usually disregarded. This results in lack of acceptance of protocols ensuring strong data consistency, which are considered too expensive and presenting poor performance. This of course, favors the adoption of protocols relaxing the consistency criteria, which present better performance but rely on user intervention or other ad-hoc mechanisms in order to bring replicas to a consistent state, when inconsistencies arise.

In distributed systems replication protocols, the properties of the communication primitives present a major concern. This results several communication primitives, ranging from reliable delivery to every destination to totally ordered and uniform delivery to every destination. The existence of such primitives eases the development of replication protocols which may rely on the communication primitives to ensure data consistency, without regarding to the data semantics.

Recently several research efforts have been developed in order to combine protocols from both communities. These efforts result in group based replication protocols [SR96, AAAS97, PGS98, SAA98, KPAS99, KA00a, KA00b, PMJPKA00, CMZ04, WK05]. The first results obtained by these protocols are encouraging, giving indicators that, in a replicated database, data consistency do not need to be sacrificed in favor of performance.

1.2 Wide-Area Networks

The underlying network topology, latency and reliability properties are factors that must be taken into consideration when developing efficient communication protocols.

In a globalized organization with multiple campus worldwide, they are connected by long distance or satellite links. This results, when compared to local area network, in latencies several times higher when using long distance links and several hundred times higher when using satellite links.

With respect to the reliability of the wide area network links, they are much more prone to failures than the ones used in the local area network infrastructure. Long distance links are usually shared by several telecommunication operators, and other infrastructures reducing the available bandwidth. Their isolation from the environment is not always the best resulting in links, being quite easily damaged even accidentally. When a problem arises with such a link, it usually means at least several hours of downtime.

Due to its cost and availability it is not always possible to have redundant long distance links which would enable operation as long as one of such link does not fails. In a local area network, even if it spreads around several buildings it is much cheaper to have redundant links connecting these facilities and ensuring that the local network is resilient to the failure of one or several of those links.

In a local area network it is usual to have multicast facilities or else low latency and high bandwidth point to point connection between every two hosts. Hosts in different local area networks of a wide-area network usually are connected using the internet, or a point-to-point communication link, with higher latency and lower bandwidth than the existent in the local area network.

These factors influence the communication protocols behavior resulting in protocols used in local area network environments defrauding users expectations in wide area network environments. For instance the changes in the point to point latency when changing from a local area network to a wide area network, require optimistic protocols to be re-evaluated in order to ensure their adequacy to such a scenario.

1.3 Partial Replication

In an organization with several branches spread world wide, there should exist data stored in the organization's information system that is relevant to all branches and information relevant to a specific branch or a subset of all branches.

Full replication of the information system data in every branch surely increases the overall system dependability, but also incurs in costs that may be considered unnecessary, relating the network bandwidth and local storage requirements at each branch.

In the cases where these costs are considered excessive, it should be considered the possibility of replicating locally at each branch the data that is only relevant locally, ensuring higher levels of dependability for local data. The other data, that is relevant to all branches, should be replicated by every replica, ensuring a dependability level able to support catastrophic events, and providing easy access to these data, as it is replicated locally.

Partial replication changes some of the assumptions upon which replication protocols have been built. These changes result mainly from the fact of dealing with incomplete information which may require additional synchronization steps among the replicas. This is specially unfortunate in wide area networks due to the higher latencies of these networks, compared to the local area network latencies.

1.4 Thesis Contributions

Replication protocols suitable for partial replication. This contribution addresses the use of partial replication in the context of the Database State Machine (DBSM) [PGS03]. We extend the Database State Machine protocol to handle partial replication while preserving its replication characteristics, namely synchronous replication and the use of the deferred update technique.

An optimistic total order protocol for wide-area networks. We propose a simple protocol which enables optimistic total order to be used in WANs with much larger transmission delays where the optimistic assumption does not normally hold. the proposal exploits local clocks and the stability of network delays to reduce the mistakes in the ordering of tentative deliveries by compensating the variability of transmission delays. This allows protocols which are based on spontaneous ordering to fulfill the optimistic assumption and thus mask the latency.

Realistic protocol evaluation through centralized simulation. This contribution addresses the difficulty of realistically evaluate the replication and communication protocols of the other contributions in various environments. Even with a complete implementation of the whole system, it is costly to setup and run multiple realistic tests with slight variations of configuration parameters. This becomes particularly evident when considering a large number of replicas and wide-area networks.

We propose a model of a replicated database server that combines simulation of the environment with real implementations of the replication and communication protocols. As these are the components responsible for the database replication, and both the database engine and the network are simulated. This allows to experiment different configuration parameters, to assess the validity of the design and implementation decisions.

1.5 Dissertation Outline

This dissertation starts with a review of replication protocols developed by the distributed systems and database communities in Chapter 2. The problem being solved by the replication protocols is the same – ensure data consistency. The review starts with the description of the five generic phases proposed in [WPS^{+}00] which may be ordered, merged, or iterated in different ways to describe the replication protocols. Afterwards, two classes of replication protocols are described

– read one write all and quorum based replication protocols. Database replication protocols based on group communication are described next, followed by the existing work on partial replication.

Chapter 3 defines the environment and provides the definitions that are required for the remaining chapters.

Chapter 4 proposes an algorithm, the *statistically estimated optimistic total order* algorithm, which tries to mask the factors responsible for the different optimistic deliveries observed by different processes, thus improving spontaneous total order in WAN. It describes the problems of total order and optimistic total order multicasts, as well as the reasons preventing spontaneous total order in WAN. Afterwards it introduces the motivation for the statistically estimated optimistic total order protocol and presents an algorithm providing optimistic delivery of messages based on a fixed-sequencer total order multicast protocol. It also describes the implementation of the statistically estimated total order protocol in a group communication toolkit, and evaluates the performance gains of the proposed protocol based on a simulated model.

Chapter 5 describes the main contributions of this thesis. It starts by revising database replication with optimistic execution, and associated database model. Afterwards, it describes the changes and refinements on the components of the database model in order to support partial replication, proposing two alternatives for the termination protocol and the trade-offs involved in the selection of each of the alternatives. It describes the protocols implementations the evaluation environment and the evaluation of the replication termination protocols using the TPC-C benchmark [TPC01].

Finally, Chapter 6 concludes this dissertation by presenting the achieved results and identifying research directions in order to complement the results obtained.

Related Publications

[1] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *IEEE International Symposium on Network Computing and Applications*. IEEE Computer Society Press, October 2001.

This paper investigates the use of partial replication in the Database State Machine approach introduced earlier for fully replicated databases. It builds on the order and atomicity properties of group communication primitives to achieve strong consistency and proposes two new abstractions: Resilient Atomic Commit and Fast Atomic Broadcast. Even with atomic broadcast, partial replication requires a termination protocol such as atomic commit to ensure transaction atomicity.

[2] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic total order in wide area networks. In *Proceedings of the 21^{st}IEEE International Symposium on Reliable Distributed Systems*, pages 190–201. IEEE Computer Society Press, October 2002.

This paper proposes a simple technique which enables the usage of optimistic delivery also in WANs with much larger transmission delays where the optimistic assumption does not normally hold. The proposal exploits local clocks and the stability of network delays to reduce the mistakes in the ordering of tentative deliveries.

[3] A. Correia Jr., A. Sousa, L. Soares, F. Moura, and R. Oliveira. Revisiting epsilon serializabilty to improve the database state machine (extended abstract). In *Proceedings of the Workshop on Dependable Distributed Data Management, SRDS (2004)*, October 2004.

This extended abstract investigates how to relax the consistency criteria of the database state machine in a controlled manner according to the Epsilon Serializability (ESR) concepts, evaluating the direct benefits in terms of performance.

[4] A. Sousa, L. Soares, A. Correia Jr., R. Oliveira, and F. Moura. Evaluating database replication in escada (position paper). In *Proceedings the Workshop on Dependable Distributed Data Management, SRDS (2004)*, October 2004.

This position paper reports the experience on the development and evaluation of group communication based database replication protocols in the ESCADA project, and points out several open issues of current research.

[5] A. Sousa, J. Pereira, L. Soares, A. Correia Jr., and F. Moura. Testing the dependability and performance of group communication based database replication protocols. In *DSN ’05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN’05)*, pages 792–801. IEEE Computer Society Press, 2005.

This paper presents a tool that combines implementations of replication and communication protocols with simulated network, database engine, and traffic generator models. This allows to evaluate replication components under realistic large scale loads in a variety of scenarios, including fault-injection, while at the same time providing global observation and control.

[6] A. Correia Jr., A. Sousa, L. Soares, J. Pereira, F. Moura, and R. Oliveira. Group-based replication of on-line transaction processing servers. In *LADC*, volume 3747 of *Lecture Notes in Computer Science*, pages 245–260. Springer, 2005.

This paper evaluates the suitability of database replication using group communication protocols for replication of On-Line Transaction Processing applications in clusters of servers and over wide area networks.

[7] A. Sousa, A. Correia Jr., F. Moura, J. Pereira, and R. Oliveira. Evaluating certification protocols in the partial database state machine. In *ARES '06: Proceedings of the First International Conference on Availability, Reliability and Security (ARES'06)*, pages 855–863. IEEE Computer Society Press, 2006.

This paper presents a detailed analysis of Partial Database State Machine (PDBSM) replication by comparing alternative partial replication protocols with full replication.

Chapter 2

Database Replication

Replication has long been a research issue either in Distributed Systems either in Databases.

In Distributed Systems, most of the research efforts have ben put on group communication protocols and on the ordering and reliability properties offered. Using the properties of the communication protocol simplifies the development of replication protocols which delegate much of its complexity to the communication protocol.

Database replication protocols focused mainly on how to improve data access concurrency. Since database replication protocols usually make no assumptions on the properties of the communication protocols, this results in complex protocols that must simultaneously deal with the communication and concurrency complexities. This is the reason why the increase in the number of the replicas may result in reduced concurrency and not in improved concurrency as expected [GHOS96].

Only recently, database replication protocols taking advantage of the properties offered by group communication protocols have emerged. This new class of protocols combines the best of both worlds, overcoming much of the limitations pointed out to existing database replication protocols, and presenting promising results.

This chapter starts with the description of the five generic phases of [WPS$^+$00], which may be combined in order to describe the existing replication protocols. Afterwards, existing replication protocols representative of two class of replication protocols – read one write all and quorum based replication protocols are described. Database replication protocols based on group communication are described next, followed by the existing work on partial replication.

2.1 Replication Protocols Classification Criteria

Replication protocols aim at ensuring consistency among a group of replicas. Despite pursuing the same goal and being conceptually similar, protocols developed in the database and distributed systems community end up very different, rendering their comparison very difficult.

In [WPS$^+$00], the authors define a set of five generic phases, that may be ordered, merged, or iterated in different ways in order to describe the replication protocols. Each of the generic phases is described as:

Request (RE) During the request phase the client submits a transaction to the system. It may send a message to one replica which in turn will send it to all other replicas during the server coordination phase (SC) or it may send the message to all replicas.

In this phase, a difference between databases and distributed systems can be observed. In database systems, clients never contact all replicas, always sending the transaction to only one of them. Two reasons justify this: the first one is that replication should be transparent to clients; and the second one is to alleviate client of the full knowledge of database internals, which is not practical for any average size database. The knowledge of the database internals resides in the replicas which will forward, when necessary, the requests to the appropriate replicas. In distributed systems, the replicas to which a request is sent makes a distinction between active and passive replication techniques[BMST93, Sch93]. In active replication, the client sends the request directly to all replicas, while in passive replication the client sends the request only to one of the replicas.

Another distinction between replication protocols in databases and distributed systems is that databases pay attention to operation semantics and dependencies between operations. Doing so database replication protocols improve concurrency, further than distributed systems protocols which do not account for operation semantics nor for dependencies among operations, only for the ordering of the operations.

Server Coordination (SC) During the server coordination phase, operations are ordered, an execution order established, and several other choices may be made by the protocols. The choices made in this phase depend on the adopted consistency criteria, and its requirements on the ordering produced by the ordering mechanisms.

In databases, operations are ordered according to data dependencies, so all operations have the same data dependencies at all replicas. This is the reason why operation semantics is so important in database replication protocols.

Distributed systems, on the other hand, are usually based on very strict notions of ordering. Ranging from causality, which is based on potential dependencies between operations, without regarding to data dependencies, to total order in which all operations are ordered regardless of being or not related.

The consistency criteria, usually adopted in replicated databases is *one-copy serializability* [BHG87]. With one-copy serializability replication should be transparent to the clients and as such the several replicas should present a behavior similar to a centralized system. In distributed systems, *sequential consistency* and *linearizability* [AW94] are used as the consistency criteria. Sequential consistency requires that all of the data operations appear to have executed atomically, in some sequential order that is consistent with the order seen at a replica. Linearizability requires that this order also preserves the global ordering of non-overlapping operations.

Execution (EX) The execution phase, usually uses locks on the data items accessed by the transactions in order to prevent concurrent transactions to update the same data items. Updates are, usually, applied to the database only during the agreement coordination phase (AC), because only after that phase the data must be made stable.

Agreement Coordination (AC) In the agreement coordination phase the replicas ensure that all or none of them have executed the transaction. This phase brings up some of the fundamental differences between distributed systems and database replication protocols.

In databases, it usually corresponds to a two phase commit protocol [BHG87], during which transaction commit or abort is decided. It is a required step, as after being ordered there are several factors that may prevent a server to execute the transaction. For example, server load, consistency constraints, interactions with local transactions, unavailability of disk space.

On the other hand in distributed systems, after the transaction is ordered it will be executed and no further coordination is required. This is the reason why in distributed systems, the factors that led a database server to abort a transaction, led the server to fail, preventing it to evolve to inconsistent states.

Response (END) In the response phase transaction's outcome is sent back to the client. Client responses can be sent: *i*) only after the transaction has been executed by all replicas, i.e., all replicas agree to commit or abort the transaction; *ii*) immediately by the replica that received and executed the transaction, which only propagates the changes to the other replicas in the server coordination phase.

In databases systems the instant where responses are sent to clients leads to: *i*) eager or synchronous replication protocols when the response is sent only after all replicas have executed the transaction; and *ii*) lazy or asynchronous replication protocols, when the client gets the result before all replicas have executed the transaction.

In distributed systems, responses are usually sent after all replicas have executed the transaction, an instant where discrepancies among replicas can not arise.

2.2 Replication Protocols

There is a plethora of replication protocols differing mainly on the necessary conditions that must hold in order to execute read or write operations. A deeper analysis of the protocols may lead to their classification along two classes: *i)* read one write all (ROWA); and *ii)* quorum based replication protocols. The first class requires write operations to be executed by all replicas, and the second one balances the difficulty of executing read and write operations by requiring those operations to be executed by a set of the replicas. These sets must ensure that *i)* any read quorum must intersect with every write quorum; and *ii)* any two write quorums must intersect.

2.2.1 Read One Write All

In the read one write all protocol (ROWA) [TGGL82], read operations are executed by the replica receiving the client requests and write operations must be executed by all replicas. Given the impossibility of executing update operations in the case of a single replica failure, a more flexible protocol, the read one write all available protocol (ROWAA) has been proposed [GSC$^+$83]. The protocol is similar to the ROWA but the definition of the available replicas is dynamic in order to reflect replica failures or network partitions.

In distributed systems, the implementations of the ROWA protocol lead to the definition of several protocols. Their differences reside in the way replicas are contacted in order to execute operations and on the mechanisms used to ensure atomic operation execution.

2.2.1.1 Active Replication

The principle of active replication, also called the state machine approach [Sch93], is that if all replicas start in the same initial state and execute the same operations in the same order, then each of them will do the same work and produce the same output. This principle implies that operations are processed in the same deterministic way.

In the active replication protocol, client operations are send to every replica, get ordered, executed by the replicas and each of the replicas replies to the client after execution. Depending on the failure model assumed, the client considers the operation executed when it receives a certain number of responses from the replicas.

According to the abstract replication protocol the active replication protocol uses the following phases:

1. (RE) The client sends the operations to the replicas using atomic broadcast [HT93].

2. (SC) During server coordination phase an order to the operations submitted by different clients is established, i.e. the order of the atomic broadcast, and operations are delivered to the replicas.

3. (EX) After delivering operations they are executed by each replica.

4. (END) All replicas send operation results back to the client.

In this protocol there is no need to agreement coordination, as operation execution is deterministic and all replicas execute the same operations in the same order, hence producing the same result.

When considering interactive transactions or transactions involving more than one operation, the request, server coordination and execution phases are executed for every operation until the commit request is issued by the client. It is worth pointing that transactions could be aborted due to concurrency control constraints, which do not happens in the single operation scenario.

2.2.1.2 Passive Replication

In passive replication or primary-backup replication [BMST93], one of the replicas, the *primary*, plays a special role: all operations requested by clients are directed to it. The primary is also responsible to execute the operations, update the other replicas and respond to the client.

According to the abstract replication protocol the primary-backup replication protocol can be described as:

1. (RE) The client sends the operation to the *primary* replica.
2. (EX) The *primary* executes it.
3. (SC) In the server coordination phase, updates produced by the execution are sent to every replica.
4. (AC) In the agreement coordination phase, an atomic commitment protocol is started, so all or none of the replicas apply the updates.
5. (END) After the termination of the atomic commitment protocol the *primary* sends to the client the result of the operation in case of success or abort otherwise.

As all transactions are executed at the primary, it is its responsibility to abort transactions due to conflicting operations. If transactions are aborted in the agreement coordination phase, it is never due to conflicting operations but unexpected events, such as disk outage or any other failures, at one or several replicas.

The required adjustments in this protocol to support interactive transaction involves a loop with the request and execution phases. When the commit request is received from the client, the protocol enters the server coordination phase and works as in the single operation case.

2.2.1.3 Semi-Active Replication

The semi-active replication protocol [PCD91], is similar to the active replication protocol, allowing for non-deterministic processing. In this protocol, non-deterministic processing is executed only by one of the replicas called the *leader* which in turn sends the result of the non-deterministic processing to the other replicas called *followers*. The resulting protocol is thus conceptually similar to state machine protocol except in non-deterministic processing where it resembles a primary-backup protocol.

The protocol can be described using the abstract replication protocol as:

1. (RE) The client sends the operations to the replicas using atomic broadcast.
2. (SC) During server coordination phase operations are ordered according to the atomic broadcast and delivered to the replicas.
3. (EX) After delivering operations they are executed by each replica, if they only require deterministic processing.
4. (SC) If there are non-deterministic choices to be made, then the *leader* makes these choices and informs the *followers*.
5. (END) All replicas send operation results back to the client.

2.2.1.4 Semi-Passive Replication

The semi-passive replication protocol [DSS98], is a replication protocol conceptually similar to the primary-backup protocol, as only one of the replicas executes the transaction while the others apply its updates.

Contrary to the passive replication protocol, in semi-passive replication client's operations are sent to all replicas, so replica failures can be made transparent to the client. Every operation is executed only by one of the replicas, selected on a per transaction basis, using the rotating coordinator paradigm [CT96]. This approach distributes transaction processing by all replicas, not overloading the primary as in the primary-backup protocol. After executing a transaction the replica starts a variant of a consensus problem, called *Consensus with Deferred Initial Values*. The result of the consensus with deferred initial values execution is the updates produced by the transaction execution. After terminating this protocol, every replica responds to the client. Using this protocol the client does not have to take care of primary failures. They are handled by the consensus with deferred initially values, which also ensures that the transaction is executed only once.

This protocol can be described using the abstract replication protocol as:

1. (RE) The client sends the operations to all replicas.
2. (SC+EX+AC) The server coordination, execution and agreement coordination phases are now merged and considered integrated in the consensus with deferred initial values. For each transaction there is a primary which is selected as the coordinator for that round of the consensus. After the selection

of the primary it executes the transaction and proposes its result as the initial value for the consensus problem. The result will be accepted by all the other replicas if it does not fail. In case of primary failure a new primary is selected and the transaction gets executed.

3. (END) After the termination of the consensus problem, all replicas apply the updates to their state and send a response back to the client.

When considering interactive transactions, all transaction operations are executed by the selected replica.

2.2.2 Quorum Based Replication Protocols

In quorum based replication protocols, before executing an operation it must be ensured that it is possible to read/write from/to a read/write quorum. In its generic form, a quorum system is a set system defined over a set of replicas. This set system holds the property that every two sets in the set system have non-empty intersection. Every set of the set system is a write quorum, meaning that a write operation must be executed by all of its members. A read-write quorum system for a set of replicas is defined over two set systems R and W, being W a quorum system and R a set system with the property that every member of R has non-empty intersection with every member of W.

A quorum based replication protocol can be described using the abstract replication protocol as:

1. (RE) the client sends its requests to one of the replicas.
2. (SC) In the server coordination phase, inquires are sent to all replicas, after which a read/write quorum containing the most up to date data is selected.
3. (EX) In the execution phase, in case of a read operation it is sent to one of the quorum members holding the most up to date version of the data. Write operations are sent to all members of the quorum.
4. (AC) The agreement coordination phase ensures that none of the quorum members have failed and that all them have executed the operation, in the case of write operations.
5. (END) After completion of the agreement coordination phase the operation result is sent back to the client.

The existing quorum based replication protocols may be classified in voting and grid quorums which are detailed in the following Sections.

2.2.2.1 Voting Quorums

Voting protocols such as the majority quorum [Tho79] and weighted voting [Gif79] are representatives of a class of quorum based replication protocols. In these, each replica has a number of votes associated to it and quorums are defined so that the number of votes necessary to form a quorum exceeds half of the total votes. Additionally a quorum must fulfill the requirements that $2wq > n$ and $rq + wq > n$, where n stands for the total of votes, wq for write quorum and rq for read quorum.

In order to tolerate replicas failure and systems dynamics, several works proposed the definition of these quorum levels dynamically according to the number of replicas [ES83, Her87, BGMS89, JM90].

Voting with witnesses [PL91, Pâr86], with ghosts [vRT88] and with bystanders [Pâr89] are several variations on voting protocols dealing with the reduction of storage required and number of replicas necessary to define a quorum in the first case and to deal with network partition failures in the other cases.

Hierarchical quorum consensus [Kum91] is a generalization of quorum consensus. It logically organizes the replicas in a multilevel tree (the root of the tree is at level 0), with leaves corresponding to the replicas and nodes to logical groups. A node at a certain level is viewed as a logical group which in turn consists of its subgroups. In order to obtain a quorum, at each level of the tree beyond the root level, a majority the nodes must be taken, and for each of those nodes a majority of its sub-nodes.

A generalization of weighted voting is the multidimensional voting presented in [AAC91]. The aim of multidimensional voting is to provide a protocol with the simplicity of the voting protocol but with the generality of quorum sets. These are showed to be more general than voting as there are quorum sets that can not be obtained by voting [GMB85]. In multidimensional voting the vote assignments to each replica and the quorums are k-dimensional vectors of non-negative integers. The quorum has a two dimension definition, a vote quorum which is a k-dimensional integer vector defining the quorum level for each dimension, and a number l, $1 \leq l \leq k$, which is the number of dimensions of vote assignments for which a quorum must be obtained.

2.2.2.2 Grid Quorums

Quorum based replication protocols focused essentially on data availability resulting in high available solutions but without noticeable performance improvement. This is a result of these protocols requiring operations to be executed by a large number of replicas, reducing the possibility of load sharing. Efforts to also improve systems performance, resulted in the definition of grid quorums [CAA92, Kum91, NW98]. In the grid protocol the N replicas are logically organized in a rectangular $m \times n$ grid. A read quorum is formed by a replica from each column of the grid and a write quorum is formed by the union of a read quorum with all the replicas of a column of the grid. The use of incomplete or hollow grids [KRS93], was suggested as an improvement on the availability of grid protocols as well as an improvement on its flexibility allowing the definition of quorums that could not be obtained with the grid protocol.

The triangular grid [EL75], is a grid protocol where replicas are arranged in a number of rows such that row i ($i > 1$) has i replicas. In it, a write quorum is the union of one complete row and one replica from every row below it. A read quorum can be either one replica from each row or a write quorum. A generalization of the triangle grid releasing the requirement that row i has i replicas has been presented as the crumbling walls protocol [PW97].

Improvements on rectangular grids availability have been proposed in [WB92, TPK95, NW98]. These protocols require quorums to be formed using paths from left to right, i.e. horizontal paths and from top to bottom, i.e. vertical paths. A read quorum is a horizontal path while a write quorum is a horizontal and a vertical path.

2.2.3 Database Replication Protocols Using Group Communication

Despite all the research efforts put on strong consistency replication protocols, their adoption has been limited. The protocols have also been severely criticized due to its inability to scale up and to its dead-lock rate of order $O(n^3)$ on the number of replicas [GHOS96].

In the distributed systems field, efficient communication protocols among a group of replicas has concentrated a lot of research efforts [Bir85, Ric93, HT93, BvR94, AMMS$^+$95a, ACBMT95, DM96, Nei96]. These efforts resulted in several implementations of group communication protocols [MSMA90, KT91, ADKM92a, ADKM92b, MAMSA94, BvR94, AMMS$^+$95a, DM96, MMSA$^+$96].

A proposal for using group communication in the processing of batch transactions [SR96], boosts the research on database replication protocols using group communication primitives. Group communication primitives have distinct properties, ranging from simple broadcast with eventual delivery to the more complex total order broadcast that ensures that all replicas deliver the same messages in the same order [HT93]. The use of different broadcast primitives results in different database replication protocols [SAA98]. The relation established between the group communication primitive and the replication protocol is such that the stronger the properties of the communication primitive, the simpler the replication protocol, either in the number of messages required either in additional protocols that must be used to ensure the correctness of the replication protocol.

The simplest replication protocol is the one using total order broadcast as the communication primitive. It is the one using fewer messages and not requiring additional protocols. It is also the protocol to which more attention has been devoted [AAAS97, PGS98, KPAS99, KA00b].

In exploiting atomic broadcast in replicated databases [AAAS97], the authors start with a naive protocol implementing a state machine [Sch93], and using a total order broadcast message per operation. The protocol then evolves to not broadcasting read operations, and finally to locally execute transactions broadcasting all transaction's operations at commit time. This results in a protocol requiring only two total order broadcast messages per transaction, or one total order broadcast message plus an atomic commitment protocol.

Starting with a protocol conceptually similar to the last one, using the deferred update principle [BHG87], and deterministic certification, a protocol requiring a single atomic broadcast message per transaction has been proposed [PGS98].

A replication protocol suited to transactions encapsulated in stored procedures has been presented in [KPAS99]. It assumes that the data is a priori partitioned in conflict classes and a transaction accesses only one of such conflict classes. This requirement has been released in [PMJPKA00], allowing transactions to access a set of conflict classes named compound conflict class. Transactions accessing the same conflict class are supposed to have high conflict probability, while transactions in different conflict classes do not conflict and can be executed concurrently. For each conflict class a master replica where update transactions are executed is defined. Update transactions are executed at the master replica despite being broadcasted to all replicas. Read only transactions can be executed everywhere using a snapshot of the data. At each replica there are a set of queues, one for each conflict class. Upon being received a transaction is queued in the respective conflict class. As soon as the transaction reaches the head of its queue it is executed if at the master replica. Otherwise it waits until the transaction's updates are received from the master replica. After being executed at the master replica, transaction's updates are broadcasted to all replicas so its changes can be installed locally and the transaction removed from its queues.

In [KA00b], the authors propose a protocol, replication with serializability, in which read operations and read-only transactions are executed locally and write operations of update transactions are deferred until commit time. At this point the transaction write-set is sent in an total order broadcast message to every replica. Upon delivery every replica requests all write locks on an atomic step, and execute the write operations after the lock request is granted. If the transaction being delivered is local and it successfully acquires all its write locks, then it will broadcast a commit message, otherwise it will broadcast an abort messge. When the commit or abort message is received, every replica commits its updates or undo them in the case of an abort, and release all locks. This protocol provides 1-copy-serializable executions, avoids distributed deadlocks, and a final two-phase commit. As optimizations to the described protocol, the authors propose two other protocols, replication with cursor stability and replication with snapshot isolation. These protocols avoid read/write conflicts thus reducing the abort rates and improve the overall performance.

The research on group based replication protocols resulted in in-core database implementations [KA00a, WK05] and a middleware implementation [CMZ04].

The Postgres-R [KA00a], implements a group based replication protocol on Postgres 6.4.2 [PSQ], a database using two phase locking for concurrency control. It requires two messages per transaction, one using total order broadcast to disseminate the transaction write sets and another using reliable broadcast send by the executing replica to commit or abort the transaction. The Postgres-R(SI) [WK05] implements the replication protocol on Postgres 7.2, a multiversion database engine. It provides snapshot-isolation as its consistency criteria [BBG$^+$95], resulting in executions that may not always be serializable.

The clustered JDBC(C-JDBC) [CMZ04], is a flexible database clustering middleware. It addresses the scalability of database clusters by dispersing the load of the database by several back-ends using RAIDb [CMZ03]. RAIDb applies to databases the same principle RAID systems apply to individual disks, hiding the back-ends from C-JDBC's clients. Its raid levels range from stripping the database, being each back-end responsible for a fragment of the database, to full replication in which every back-end holds a full copy of the database. In order to improve C-JDBC's scalability and eliminate its single point of failure two alternatives are proposed. Horizontal scalability, obtained by replicating the C-JDBC controller, resulting also in an improvement on system fault-tolerance. Vertical scalability, obtained cascading C-JDBC controllers, allowing the database replication layer to adapt to the requirements of its applications. A solution improving both scalability and fault-tolerance is the combination of both horizontal and vertical scalability proposals.

The performance evaluation of the group communication based replication protocols has always been present [HAA99, KA00a, JPPMKA02, AT02, ADMA$^+$02, CMZ04, WK05]. The first proposed protocols using group communication primitives have been evaluated using simulation in [HAA99]. As expected it reveals the protocol broadcasting all updates in a single message as having lower execution latency. The other results have been obtained in several scenarios and using distinct applications. One common aspect to all these performance evaluations is the fact that all of them use real implementations of the group communication and replication protocols. As the baseline for the evaluation, [CMZ04, WK05] use a non-replicated database, [KA00a, JPPMKA02] use a replicated database with distributed locking and [AT02, ADMA$^+$02] use a replicated database using two phase commit protocols. Additionally in [ADMA$^+$02] the protocols were evaluated in wide area networks.

In all evaluations of group based replication protocols, improvements in overall system performance have been obtained indicating that it seems an effective way of improving systems performance without sacrificing its correctness.

2.2.4 Partial Database Replication

The replication protocols described so far only consider full replication scenarios. A question arising is whether they are also applicable in partial replication scenarios. Initial works in partial replication [Alo97], argue that results obtained for fully replicated databases are not directly applicable in partially replicated databases. This happens because some discrepancies among the orders obtained by different replicas may arise, leading to inconsistencies among them. This same problem, of discrepancies between serialization order and temporal precedence, arises in federated and multi-database systems. In [Alo97], order preserving serializability [BBG89] is presented as a sufficient condition for obtaining partial replication protocols based on group communication.

The use of epidemic protocols for partial database replication has been proposed in [HAA02]. The replication protocol uses epidemic multicast of database logs in order to disseminate and order transactions. The logs produced by a transaction are sent to every replica independently of it holding or not a replica of the modified data. The difference between this protocol and its full replication counterpart [HAA00], is that a replica only updates data items referred in the log for which it holds a replica. When a transaction tries to access a data item not replicated locally, it places an entry in the log requesting it, generating a new replica, and keeps the replica updated as long as it finds it useful.

2.3 Total Order protocols

Total order protocols are used as a building block of some of the replication protocols describe earlier in Section 2.2. In fact the group based replication protocols of Section 2.2.3, using total order in the transaction's dissemination are the simpler ones and also the ones requiring the smaller number of messages to be exchanged by the replicas.

In [DSU04] the authors studied total order protocols and proposed a classification of the total order protocols according to the mechanism used to order the messages. They define five classes of ordering mechanisms: fixed sequencer, moving sequencer, privilege based, communication history and destinations agreement. Each of these classes represents a set of protocols that are likely to exhibit similar behavior.

In the fixed sequencer total order multicast protocols, the process sending a message sends it to all the destination processes and to the sequencer. After receiving the message the sequencer establishes the message order and multicasts the order to all destinations.

The moving sequencer total order multicast protocols are conceptually similar to the fixed sequencer ones. The difference is that the initial message is multicasted to all sequencers instead of to the fixed sequencer. When receiving the message, only one of the sequencers orders it and multicasts the order to all of its destinations, informing all the other sequencers of the message it ordered.

Since these classes of protocols are conceptually similar, we merged them into a single class named of sequencer based total order protocols.

The privilege based total order multicast protocols assume the existence of a token. The token circulates among all processes and grants to its owner the permission to multicast and order messages. Messages multicasted while the process is not the token holder are queued. When receiving the token, the process orders all messages in its queue and multicasts them. Afterwards it sends the token along with the current ordering version to the next token holder.

Regarding the communication history based total order multicast protocols, in the ones using causal history, each process has a *timestamp* which it increases by 1 when sending a message, attaching it to the message being sent. In the reception of a message, the process timestamp is increased by 1 if the process timestamp is higher than the timestamp of the received message. If the timestamp of the received message is higher than the local timestamp, then the message timestamp is increased by 1 and becomes the process timestamp. Messages are delivered according to their timestamp and messages with the same timestamp are ordered according to to the identifier of its sender. These protocols can only deliver a message m after receiving from every process, a message that was multicasted after the reception of m, or that is concurrent with m, i.e., that has the same timestamp as m.

The destinations agreement total order multicast protocols require that all processes agree on the messages order. Their agreement can be on the message order, on a message set or on the proposed message order. In the protocols using agreement on the message order, the sender multicasts the message to be ordered to all processes, which will assign a local timestamp to it and multicast this timestamp to all processes. When the local timestamp is received from all processes the message global timestamp is established and messages are delivered according to the global timestamp.

From the algorithms description, the communication patterns, as well as the expected execution latency of each protocol class can be established. The communication patterns and expected latencies of the protocols execution are presented in Table 2.1, where n and m are number of processes, with $m < n$, k is the number of messages queued and l is the point to point latency.

	Sequencer	Privilege based	Communication history	Destinations agreement
Comm. Pattern	$n + n$	$1 + (\frac{n}{k} \vert n)$	$n + m \times n$	$n + n \times n$
Latency[a]	$l + l$	$\frac{n}{2}l$	l	$l + l$
Latency[b]	$2l + \frac{6l}{n}$	$\frac{n+6}{2}l$	$4l$	$4l + 4l$

[a] Assumes that point to point latency between any two processes is l.

[b] Assumes that point to point latency between $n - 1$ of the processes is l, and that the remaining one has a point to point latency of $4l$.

Table 2.1: Total order protocols communication pattern and latency.

In terms of communication pattern, the communication history is the protocol requiring fewer messages when all processes send messages at the same rate, requiring a single multicast message to order a message (i.e., $m = 0$). It may require some additional messages (i.e., $m \neq 0$), when some process, m, send messages less frequently than the others, in order to avoid latency increases.

The privilege based protocol requires a point to point message from the token holder to the next token holder and *i)* a multicast message from the token holder to multicast and order all the messages in the queue; or *ii)* k multicast messages to multicast and order each of the k messages in its queue. When several messages are multicasted and ordered in a single multicast message, the privilege based is the protocol requiring fewer messages to order a set of messages.

The sequencer based protocol requires two multicast messages to order a message, i.e., a multicast message from the sender to all destinations and a multicast message from the sequencer to all destinations.

The destinations agreement protocol requires a multicast message from the sender to all destinations followed by a multicast message from each destination to all the destinations to send the ordering info.

Regarding the protocols latency, in a scenario where the processes point to point latencies are equal, the communication history is the one with the lowest latency, in a optimal scenario where all processes are sending the same number of messages and at the same message rate. In this scenario, the protocol latency equals the point-to-point latency.

The sequencer based and destinations agreement protocols present similar latencies which are twice as large as the point to point latency. In the sequencer based it is the latency of a message from the sender to the sequencer plus the latency from the sequencer to all destinations. In the destinations agreement the latency is the sum of the latency of the message from the sender to all destinations plus the latency of the message sent from each of the destinations.

In the privilege based protocol, the latency of the protocol depends on the instant a message is sent. if it is sent immediately after the token is passed away then the latency will be equal to the time it takes to the token to reach this sender again. If the message is sent immediately before being the token holder, then the latency will be close to 0. Considering the middle case, i.e. a scenario in which messages are sent at a constant rate, then the latency will be equal to $\frac{(n)}{2}l$.

Considering a scenario in which a process point to point latency is four times higher than the other processes point to point latency, this single process with higher latency will affect the latency of all protocols. The sequencer based protocols are the ones which the smallest increase in the protocol latency. In the privilege based protocols the token period will be increased by the latency of the process with higher point to point latency, and the communication history and destinations agreement protocols will observe a latency increment proportional to the highest point to point latency among every two processes.

2.4 Summary

Replication has long been a research issue in databases and distributed systems. However, few commercial databases integrate results derived from research. Usually replicated databases do not support synchronous replication, i.e. replication protocols ensuring one copy serilizability. Distributed systems based on group communication can be found in military environments such as the AEGIS warship, in trading systems such as the New York and Swiss stock exchange, in air traffic control, factory process control and telephony.

In distributed systems, communication complexities such as reliability and ordering guaranties are hidden from the application. Application developers use the communication primitives provided, usually by a group communication toolkit. This enable them to concentrate in the application development rather than on ensuring the properties required for the communication primitives.

In databases, usually there is no separation between the database and the communication layer. The use of a generic communication layer means that it does not provide reliability or ordering guaranties, so the programmer must implement them and integrate them with the database. Doing so, the programmer must concentrate in its application but also in every other aspect related to communication, making it difficult to implement efficiently the replication strategies proposed in the literature, due to the involved complexity.

The work presented in [JPPMAK03], studied several quorum based replication protocols and conclude that the ROWA protocol is the adequate one for most situations. It is also the protocol implemented by most of the group based replication protocols. The group based replication protocols presenting better performance results are the ones using optimistic execution and total order multicast to disseminate and order transactions.

The total order protocols have also been a research issue for a long time and a recent survey [DSU04] classified the existing protocols according to the mechanism used to order messages, which resulted in 5 different classes of protocols. The observation of a simple algorithm of each class allowed to establish the communication pattern and protocol execution latency of each class.

The analysis of the communication pattern showed that in a system with n processes, the privilege based protocols require $n+1$ messages to deliver in total order a single or a set of messages. The communication history protocols require, in an ideal scenario, n messages to deliver in total order each message. The sequencer based protocols requires $2n$ messages and the destinations agreement protocols require $n+n^2$ messages to deliver in total order each message. This results in that if using the communication pattern as the selection criteria, then privilege based or communication history protocols should be selected.

The analysis of the protocol execution latency revealed that communication history and destinations agreement protocols execution latency is directly related to the point to point latency of a single process exhibiting high latency. The privilege based protocols are additionally affected by the number of processes. The only class of protocols whose execution latencies are not directly influenced by the existence of a single process with higher point to point latency is the sequencer based class of protocols. In this case the only messages which have higher ordering latency are the ones sent by the process with higher latency. From this it results than when performance is the selection criteria, and there is variable point to point latency among the processes, then the sequencer based protocols should be selected.

Chapter 3

Models and Definitions

3.1 Distributed System Model

The considered distributed system is composed of two completely connected sets $S = \{s_1, ..., s_n\}$ and $C = \{c_1, ..., c_m\}$, respectively of database sites or processes, and client sites.

Database sites communicate through message passing (i.e., no shared memory). Communication is done through a fully connected network in which, reliable and totally ordered broadcast primitives are available [HT93]. Reliable broadcast, totally ordered broadcast and delivery of messages are represented by steps *rb-cast(m)*, *abcast(m)* and *bdel(m)*. Uniform agreement is assumed to hold [HT93]: A message delivered by any database site, is eventually delivered to all correct database sites.

A database site is *correct* if, in response to inputs, it behaves in a manner consistent with its specification [Cri91]. A failure occurs when the database site behavior is different from its specification.

If, after a first failure to behave according to its specification, a database site omits to produce output to subsequent inputs until its restart, the database site is said to suffer a *crash failure* [Cri91]. In the assumed system model, database sites fail only by crashing (i.e., no Byzantine failures [Cri91]), and do not recover, thus stopping to execute database operations, or broadcast or deliver further messages. A crash is modeled as the repeated execution of event CRASH. It is assumed that scheduling of events in processes is fair: No event is forever ready to be executed without in fact being executed.

The availability of totally ordered broadcast implicitly assumes that consensus is solvable in the assumed system model [FLP85]. It is not explicitly explained how this is achieved or otherwise make use of any assumptions besides an asynchronous system model where sites fail only by crashing and do not recover.

3.2 Databases and Transactions

A database is defined using the page model [GG02]: A database $\Gamma = \{x, y, z, \ldots\}$ is a collection of named data items which have a value. The combined values of the data items at any given instant is the database state. There are no assumptions on the granularity of data items.

A database system (DBS) is a collection of hardware and software modules that support commands to access the database, called database operations (or simply operations). Data items are manipulated by read and write operations. A read operation on data item x, depicted as $r(x)$, returns the value stored in data item x, and a write operation, depicted as $w(x)$, changes the value of data item x.

The DBS executes each operation atomically, i.e., it behaves as if operations execution is sequential. However, typically it will execute operations concurrently. That is, there may be times when it is executing more than one operation at once. However, even if it executes operations concurrently, the final effect must be the same as some sequential execution.

The DBS also supports transactions whose semantics are ruled by the ACID properties [GR93]:

Atomicity A transaction's changes to the state are atomic: either all happen or none happen.

Consistency A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state.

Isolation Even though transactions execute concurrently, it appears to each transaction, t, that others executed either before t or after t, but not both.

Durability Once a transaction completes successfully (commits), its changes to the state survive failures.

A transaction t is a partial order on a set of operations of the form $r(x)$ or $w(x)$, where $x \in \Gamma$, with an initial operation (start transaction) depicted as b_t and final operation, commit depicted as c_t or abort depicted as a_t, depending on the transaction termination status. I.e., if a transaction ends with commit then all of its updates have been applied to the database. If it ends with abort then none of its update operations are visible to other transactions. Transaction reads and writes, as well as multiple writes applied to the same data item are ordered. A transaction with m operations is a pair $t = (op, <_t)$ such that:

- $\forall_i 1 < i < m, op_t^i \in \{r(x), w(x)\}$
- $op_t^i <_t op_t^j$ iff $i < j$
- $op_t^1 = b_t$
- $op_t^m = c_t | op_t^m = a_t$

The result of executing a transaction t is a sequence of reads and writes of data items. The read set of the transaction, denoted by $RS(t)$, is the set of identifiers of the data items read by t. The write set, denoted $WS(t)$, is the set of identifiers of the data items written by t. The set of data items written by t, its write values, is denoted $WV(t)$.

A partially replicated database is a set of DBSs, each of them called a database site, holding a partial copy of the database data items. For each data item $x \in \Gamma$, it is assumed that there is at least one correct database site that stores x. For each database site $s \in S$, $Items(s)$ is defined as the set of data items replicated in s; the set of all database sites replicating a data item $x \in \Gamma$ is denoted by $Sites(x)$. Given a transaction t, $RS(t).s$ and $WS(t).s$ identifies the data items read or written, respectively, by t and replicated in a particular database site s.

Transactions execution is formalized by schedules and histories [GG02]. A schedule is a prefix of a history, which defines in what order a set of transactions $T = \{t_1, \ldots, t_n\}$ are executed in a database. Each $t_i \in T$ has the form $t_i = (op_i, <_i)$, with op_i denoting the set of operations of t_i and $<_i$ denoting their ordering $1 \leq i \leq n$. A history for T is a pair $s = (op(s), <_s)$ such that:

- $op(s) \subseteq \cup_{i=1}^n op_i \bigcup \cup_{i=1}^n \{b_i, a_i, c_i\}$ and $\cup_{i=1}^n op_i \subseteq op(s)$, i.e., s consists of the union of the operations from the given transactions plus a starting operation, b_i, plus a termination operation which is either a c_i (commit) or an a_i (abort) for each $t_i \in T$;
- $(\forall_i 1 \leq i \leq n)\ c_i \in op(s) \iff a_i \notin op(s)$, i.e., for each transaction, there is either a commit or abort in s, but not both;
- $\cup_{i=1}^n <_i \subseteq <_s$, i.e., all transaction orders are contained in the partial order given by s;

- $(\forall_i, 1 < i \leq n)(\forall_{p \in op_i})\ p <_s b_i$, i.e., the start operation always appears as the first step of a transaction.

- $(\forall_i, 1 \leq i < n)(\forall_{p \in op_i})\ p <_s a_i \vee p <_s c_i$, i.e., the commit or abort operation always appears as the last step of a transaction;

- every pair of operations $p, q \in op(s)$ from distinct transactions that access the same data item and have at least one write operation among them is ordered in s in such a way that either $p <_s q$ or $q <_s p$.

A history s is serial if for any two transactions t_i and t_j in it, where $i \neq j$, all operations from t_i are ordered in s before all operations from t_j or vice-versa.

In a schedule s, conflicts between transactions t and t', such that $t \neq t'$ and both belong to the schedule, are characterized as:

- two data operations $p \in t, q \in t'$ are in conflict in s if they access the same data item and at least one of them is a write operation, i.e., $(p = r(x) \wedge q = w(x)) \vee (p = w(x) \wedge q = r(x)) \vee (p = w(x) \wedge q = w(x))$

- $conf(s) \equiv \{(p, q) | p, q$ are in conflict in s and $p <_s q\}$, is called the conflict relation of s.

Two schedules s and s' are conflict equivalent, denoted $s \approx_c s'$, if they have the same operations and the same conflict relations, i.e., if:

- $op(s) = op(s')$ and

- $conf(s) = conf(s')$

A history s is conflict-serializable if there exists a serial history s' such that $s \approx_c s'$. The class of all conflict-serializable histories is denoted CSR.

The conflict graph $G(s) = (V, E)$ of a schedule s is defined by:

- $V = c_t$

- $(t, t') \in E \iff t \neq t' \wedge (\exists p \in t)(\exists q \in t')(p, q) \in conf(s)$

I.e., the vertices of the graph are the committed transactions in s and there is an edge between two committed transactions if there are conflicting operations between the transactions.

The serializability theorem states that a history s belongs to class CSR iff its conflict graph $G(s)$ is acyclic.

In a database, the existence of several versions of a data item is a method to improve concurrency among transactions, allowing executions otherwise impossible. For instance, the existence of several versions of a data item allows for transactions to keep reading a data item from an older version despite a transaction generating a new version of that data item have been committed. The improvement in concurrency results from the fact of the reading transaction continue its execution, instead of being aborted as in the case where there was only a version of the data item.

The multiversion model which considers several versions of a data item is attractive for non-replicated databases as it improves concurrency, but also allows to model replicated databases, in which, each replica is considered as a version of the database.

The multiversion model depends on a version function h defined as: Let s be a history with initialization transaction t_0 and final transaction t_∞. A version function for s is a function h, which associates with each read step of s a previous write step on the same data item, and which is the identity on write steps.

In the multiversion model, a history, i.e., a multiversion history on a set $T = \{t_1, \ldots, t_n\}$ is defined as is a pair $m = (op(m), <_m)$ where $<_m$ is an order on $op(m)$ and:

- $op(m) = h(\cup_{i=1}^{n}(op_{t_i}))$ for some version function h;
- For all $t \in T$ and all operations $p, q \in op(t)$ the following holds: $p <_t q \Rightarrow h(p) <_m h(q)$;
- if $h(r_j(x)) = r_j(x_i), i \neq j$, and c_j is in m, then c_i is in m and $c_i <_m c_j$;

A multiversion schedule is a prefix of a multiversion history.

If a multiversion schedule version function maps each read step to the last preceding write step on the same data item, then this schedule is called a monoversion schedule.

In a multiversion schedule m, a multiversion conflict is a pair of steps $r_i(x_j)$ and $w_k(x_k)$ such that $r_i(x_j) <_m w_k(x_k)$.

A multiversion history m is multiversion conflict-serializable if there is a serial monoversion history for the same set of transactions in which all pairs of operations in multiversion conflict occur in the same order as in m. The class of all multiversion conflict serializable histories is denoted $MCSR$.

The multiversion conflict graph of a multiversion schedule m is a graph that has the transactions of m as its nodes and an edge from t_i to t_k if there are steps $r_i(x_j)$ and $w_k(x_k)$ for the same data item x in m such that $r_i(x_j) <_m w_k(x_k)$.

A multiversion history is $MCSR$ iff its multiversion conflict graph is acyclic.

3.3 Fault-tolerant Communication Primitives

Process communication constitutes one of the basic building blocks of a distributed system. The guaranties provided by the communication layer cannot be disregarded as they also play an important role on application development. Depending on the application different scenarios on message reliability and ordering might be necessary. This section recalls the definition of some communication primitives defined in [HT93].

3.3.1 Reliable Broadcast

Reliable broadcast is the weakest type of fault-tolerant communication primitive considered. It is a broadcast primitive that satisfies the following three properties:

- *validity:* If a correct process broadcasts a message m, then it eventually delivers m;
- *Agreement:* If a correct process delivers a message m, then all correct processes eventually deliver m;
- *Integrity:* For any message m, every correct process delivers m at most once, and only if m was previously broadcast by $sender(m)$.

3.3.2 Atomic Broadcast

When all messages must be delivered in the same order by all the processes this characterizes an Atomic Broadcast, a Reliable broadcast with the following total order property:

- *Total Order:* If correct processes p and q both deliver messages m and m', the p delivers m before m' if and only if q delivers m before m'.

Chapter 4

Optimistic Total Order in Wide Area Networks

Total order multicast greatly simplifies the implementation of fault-tolerant services using the replicated state machine approach [Sch93]. By ensuring that deterministic replicas handle the very same sequence of requests from clients, it is ensured that their state is kept consistent and the interaction with clients is serializable [GS97b].

Implementation of total order multicast is however more costly than other forms of multicast. The establishment of a total order has an unavoidable impact on latency. For instance, in a sequencer based protocol [BSS91, KT91] all processes (except the sequencer itself) have to wait for the message to reach the sequencer and for the sequence number to travel back before the message can be delivered.

Protocols based on causal history [Lam78, PBS89, EMS95] can provide latency proportional to the inter-arrival delay of each sender and thus lower latency than sequencer based protocols. However, when each sender has a large inter-arrival time and low latency is desired, this requires the introduction of additional control messages. This is especially unfortunate in large groups and in wide area networks (WAN) with limited bandwidth links.

In most protocols, based on a sequencer [BSS91, KT91] or on consensus [CT96, Anc96], the total order is given by the spontaneous ordering of messages as observed by some process. In addition, in local area networks (LAN) it can be observed that the spontaneous order of messages is often the same in all processes. Disclosing the spontaneous ordering of messages to the client application, compensates part of the total order multicast latency by allowing the computation to proceed in parallel with the ordering protocol [KPAS99]. Later, when the total order is established and if it confirms the spontaneous order, the application can immediately use the results of the *optimistic* computation. If not, it must undo the effects of the computation and restart it using the correct ordering.

The effectiveness of the technique rests on the assumption that a large share of correctly ordered optimistic deliveries offsets the cost of undoing the effects of mistakes. While this is the case for LAN, in WAN loopback optimization, packet loss, network topology and routing policies are responsible for an increase in the number of mistakes. This is unfortunate as this makes optimistic delivery useful only in LAN where the latency is much less of a problem than in WAN.

This chapter proposes an algorithm, the *statistically estimated optimistic total order* algorithm, which tries to mask the factors responsible for the different optimistic deliveries observed by different processes, thus improving spontaneous total order in WAN.

This chapter is organized as follows: the next section recalls the problems of total order and optimistic total order multicasts, as well as the reasons preventing spontaneous total order in WAN. Section 4.2 introduces the motivation for the statistically estimated optimistic total order protocol and presents an algorithm providing optimistic delivery of messages based on a fixed-sequencer total order multicast protocol. Section 4.3 describes the implementation of the statistically estimated total order protocol in a group communication toolkit. Section 4.4 evaluates the performance gains of the proposed approach based on a simulated model. Finally, Section 4.5 summarizes the results obtained by the statistically estimated optimistic total order protocol and discusses its applicability.

4.1 Optimistic Total Order Multicast

4.1.1 Total Order Multicast

Total order multicast ensures that no pair of messages is delivered to distinct destination processes in different order.

Formally, total order multicast is defined by primitives *to-multicast*(m) and *to-deliver*(m), and satisfies the following properties [HT93]:

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

Agreement. If a correct process to-delivers a message m, then all correct process eventually to-delivers m.

Integrity. For any message m, every correct process to-delivers m at most once, and only if m was previously to-multicast by the sender of m.

Total Order. If correct processes p and q both to-deliver messages m and m', then p deliver m before m' if and only if q delivers m before m'.

Total order multicast has been shown to be equivalent to the generic agreement problem of consensus [CT96] and therefore assumed to be solvable in our assumed system model (Chapter 3). In some protocols, consensus is explicitly invoked to decide the message sequence [CT96, Anc96]. In others, consensus is implicit in a group membership service [BSS91, HLvR99, EMS95].

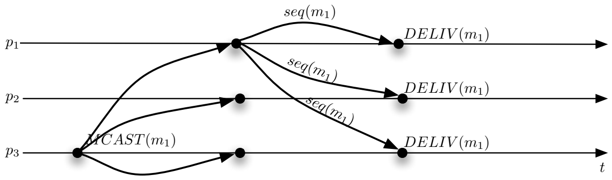


Figure 4.1: Sequencer based total order protocol.

There is a plethora of total order protocols for asynchronous message passing systems which can be classified according to several criteria [DSU04]. Some order the messages while disseminating them [AMMS^{+}95b, ACM95, HLvR99]. Others, take advantage of an existing unordered multicast protocol [BSS91, KT91, CT96] and work in two stages: First, messages are disseminated using a reliable multicast protocol. Then, an ordering protocol is run to decide which is the correct delivery sequence of buffered messages.

An example of a protocol often used in group communication toolkits is the one based on a sequencer [BSS91, KT91], which uses consensus implicitly in the view-synchronous reliable multicast protocol used to disseminate messages previous to ordering them. As depicted in Figure 4.1, a data message is disseminated using an unordered reliable multicast primitive. Upon reception (depicted as a solid dot), the message is buffered until a sequence number for it is obtained. A single process (p_1 in the example) is designated as the sequencer; it increments a counter and multicasts its value along with the original message identification to all receivers as a control message. Data messages are then delivered according to the sequence numbers. A group membership protocol is used to ensure that for any given data message there is exactly one active sequencer.

Besides being a very simple protocol, it performs very well, especially in networks with limited bandwidth or in large groups with large and variable message inter-arrival times: it requires at most a single additional control message for each data message and any message can always be delivered after two successive message transmission delays. The basic protocol is also easily modified to cope with higher message throughput by batching sequence numbers for several messages in a single one [BSS91], reducing the number of control messages at the expense of a higher latency.

4.1.2 Optimistic Total Order

A reliable multicast protocol can deliver a message after a single transmission delay from the originator to the receiver. This contrasts with the latency of totally ordered multicast which is twice as large when using a sequencer based protocol, or proportional to the message inter-arrival delay in protocols using causal history. However:

- Some protocols, such as the sequencer based, produce an ordering which is the spontaneous ordering observed by some process.

- In LAN, it can be observed that the spontaneous ordering of message reception of all processes is often the same and therefore, the same as the final ordering decided by the sequencer.

The optimistic total order multicast protocol [PS03] takes these observations into consideration to improve average delivery latency of a consensus based total order protocol.

Further latency improvements can be obtained if the application itself can take advantage of a tentatively ordered delivery. This is called optimistic delivery [PS03, SPOM01] as it comes from the optimistic assumption that reliable multicast orders messages spontaneously. It also implies that eventually an authoritative total order is determined, leading to a confirmation or correction of the previously used delivery order. To the interval between the optimistic delivery and the authoritative delivery we call *optimistic window*. It is during this interval that the application can optimistically do some processing in advance.

To define optimistic total order multicast two different delivery primitives are used. An optimistic *opt-deliver*(m) primitive that delivers messages in a tentative order and a final *fnl-deliver*(m) primitive that delivers messages in their final, or authoritative, order. Optimistic total order multicast satisfies the following properties [SPOM01]:

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

Agreement: If a correct process fnl-delivers a message m, then every correct process eventually fnl-delivers m.

Integrity: For every message m, every process:

- opt-delivers m only if m was previously multicast, and;
- every process fnl-delivers m only once, and only if m was previously multicast.

Local Order: No process opt-delivers a message m after having fnl-delivered m.

Total Order: If two processes fnl-deliver two messages, then they do so in the same order.

Notice that if the optimistic ordering turns out to be wrong, the application has to undo the effect of any processing it might have done. Therefore, the net advantage of optimistic delivery depends on the balance between the cost of a mistake and the ratio of correctly ordered optimistic deliveries. When the optimistic delivery is wrong, there is a performance penalty: The processing resources used have been wasted.

The tradeoff is thus similar to the one involved in the design of cache memories. However, the protocol designer has no possibility to reduce the cost of a mistake, as this depends solely on the application. The only option is thus to try to maximize the amount of messages which are delivered early but correctly ordered.

4.2 Statistically Estimated Optimistic Total Order

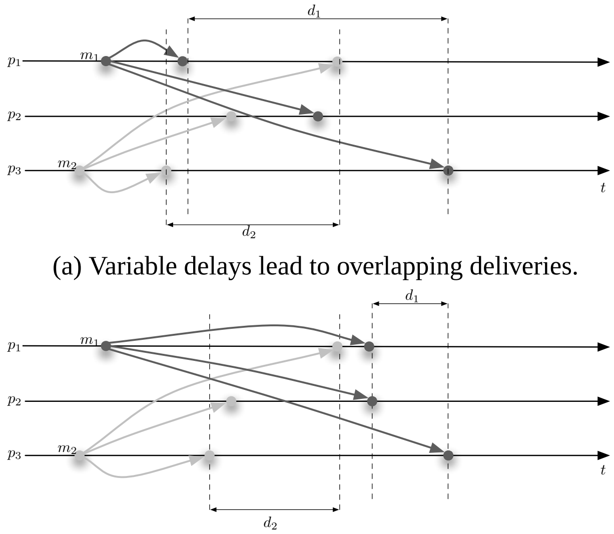
4.2.1 Obstacles to Spontaneous Total Order

A high ratio of spontaneously totally ordered messages which results in good performance of optimistic applications is not trivially achieved, especially in WAN. One reason for this is the loopback optimization in the operating system's network stack. Noticing that the outgoing packet is also to be delivered locally, the operating system may use loopback at lower layers of the protocol stack and immediately queue the message for delivery. This allows self messages to be delivered in advance of messages from other senders which have reached the network first.

Another reason for out of order delivery lies in the network itself. Although not frequent, there is a possibility that messages are lost by some but not all destinations. A reliable multicast protocol detects the occurrence and issues a retransmission. However, the delay introduced opens up the possibility of other messages being successfully transmitted while retransmission is being performed.

An additional issue is the complexity of the network topology. Different messages can be routed by different paths, being therefore subject to different queuing delays or even to being dropped by congested routers. This is especially relevant when there are multiple senders. Receivers which are nearer, in terms of hops, to one of them will receive its messages first. Receivers which are nearer of another will possibly receive messages ordered differently.

Notice however that bad spontaneous order in WAN is not attributable to the large network delays themselves, but to the fact that the delays to different destinations are likely to be different, often by two orders of magnitude. Consider Figure 4.2(a). Messages m_1 and m_2 are multicast to three different processes, including the senders themselves. The time taken to transmit each message varies with the recipient, for instance, transmission to the sender itself (typically hundreds of microseconds by loopback) takes less time than transmission to other processes (typically up to tens of milliseconds over a long distance link). The result is that process p_1 spontaneously orders message m_1 first while p_2 and p_3 deliver m_2 first.



(a) Variable delays lead to overlapping deliveries.

(b) Similar delays avoids overlapping deliveries.

Figure 4.2: Transmission delays and spontaneous total order.

Figure 4.2(b) shows a similar example where message transmission delays are longer but where is it more likely that messages are delivered by all processes in the same order. What actually matters is the difference between transmission delays to different processes depicted as d_1 and d_2. Larger values for d_1 and d_2 mean that there is a higher probability of transmission overlapping and thus of different delivery order.

4.2.2 Delay Compensation

A network exhibiting identical transmission delays with low variance among any pair of processes would enable spontaneous total ordering of messages. This observation leads to the intuition underlying the proposed protocol: given the magnitude of the latency introduced by total order protocols it should be possible, by judiciously scheduling the delivery of messages, to reduce the differences among transmission delays and *produce* an optimistic order which is likely to match the authoritative total order. As an example, notice that Figure 4.2(a) can be transformed into Figure 4.2(b) simply by delaying the deliveries of message m_1, leading to a total order of m_2 and m_1;

What remains to be established is how to determine the correct delays to introduce to each message such that the likelihood of matching the authoritative total order is improved. The challenge is to do this with minimal overhead, both in terms of messages exchanged as well as computational effort. In addition, by introducing delays this technique increases the average latency of optimistic delivery. This must therefore be minimized and compensated by the higher share of correctly ordered optimistic deliveries.

Notice that in a WAN this cannot ever replace a total order algorithm: Transmission delays cannot be precisely estimated, some uncertainty exists and thus it is likely that some messages are delivered out of order [Pax97]. On the other hand, if the only modification to the original sequencer algorithm is the introduction of finite delays, its correctness in an asynchronous system model is unaffected. Therefore, by reusing an algorithm known to be correct in the asynchronous system model, the robustness of the solution is ensured [OPS01]. Timing assumptions, namely on the stability of transmission delays as measured by a process local clock are used only to improve the performance.

4.2.2.1 Relatively Equidistant Receivers

As the basis for the protocol, it is considered a fixed-sequencer total order multicast algorithm as described in Section 4.1.1. It is assumed that the total order of messages is based on the spontaneous ordering of messages as seen by the sequencer.

The different orders seen by a process p between the messages it delivers optimistically and those that it delivers authoritatively reflect the *relative* differences between the communication delays from the senders to p and to the sequencer. For simplicity these communication delays are treated as "distances" between processes (more precisely, as directed distances, as the distance from p to q can be different from that of q to p). If, through the introduction of artificial delays, each process p and the sequencer are set as relatively equidistant receivers with respect to all other processes, then the order in which p delivers messages optimistically will be that of the sequencer and therefore will match the authoritative order.

The way to increase the distance from q to p is to delay the optimistic delivery of messages from q at p. This means that when p needs to get q closer either p reduces the delay it might be imposing to the messages from q, or p has to stand back from all other processes by delaying the optimistic delivery of messages from these processes. This is the basic mechanism of the algorithm. It is simple and independently managed at each process, e.g., the adjustment of the distance between p and q is independent from that between q and p.

Two particular cases however require special attention. One is the fact that any process is usually closer to itself than from the sequencer and thus it will have to distance from itself. This case is simple, each process will delay the optimistic delivery of its own messages such that "it distances from itself" as it distances from the sequencer. The other case regards the sequencer itself. While, as any other process, the sequencer is closer to itself than to the other processes. The distance to the sequencer does not apply here and the order of optimistic delivery trivially matches that of the authoritative's. However, it is required, as happens with the other processes, that the sequencer "distances from itself" by delaying the optimistic delivery of its own messages.[1] The reason for this is that unless the sequencer delays the optimistic delivery of its own messages, the optimistic and authoritative delivery of its messages will always occur almost simultaneously. This is true at the sequencer process itself as well as in any other process and, as exemplified in the next section, it would eventually force the same phenomenon in the messages of the other processes. The problem of delaying the optimistic delivery of the sequencer's messages is that it also delays their authoritative delivery.

[1] The delay imposed on the optimistic delivery of sequencer's own messages may impact all other processes and be responsible for an increase in the overall latency, as sequencer's messages are only ordered after being optimistically delivered.

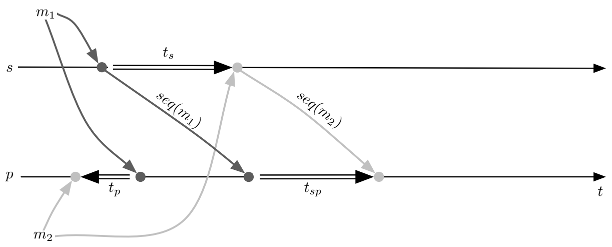


Figure 4.3: Calculation of relative equidistance.

4.2.2.2 Distance Calculation

Consider the scenario depicted in Figure 4.3. Let s represent the sequencer's timeline and p that of some process p. Message m_1 is multicast by a process p_1 (not shown) and message m_2 is multicast by another process p_2 (also not shown). Both the sequencer s and a second process p receive m_1 and m_2 as shown. Upon reception they are ordered by s, which assigns them sequence numbers and delivers them immediately. The authoritative order of the messages becomes m_1, m_2 as this was the spontaneous order seen by s. In contrast, process p can only make an optimistic guess about the final relative order of m_1 and m_2, and in this situation it would have mistakenly predicted the delivery of m_2 before m_1. The final order is known only upon the reception of the sequence numbers from s.

As soon as it receives the sequence numbers for both messages, p becomes aware that its relative distances to p_1 and p_2 are different from those of s, because it has received the messages in the inverse order. If it had delayed the optimistic delivery of m_2 until after the reception of m_1, it would have compensated its relative distance from the senders with respect to that of the sequencer and matched the authoritative order.

Although any sufficiently large delay imposed on m_2 by p would correctly order it relatively to m_1, a correct prediction of the final order by p requires an evaluation of relative distances to senders to s and to p, enabling an optimal delay to be introduced. Notice that the delay should not be so large that it causes m_2 to be misordered with a future message m_3 that arrives to all processes after both m_1 and m_2.

Explicit estimation of distances among all processes is not required. A better approach is to directly determine optimal delays to be introduced prior to optimistic delivery by observing that:

- If the relative distance of p and s is the same with respect to senders of m_1 and m_2 and each message is multicast simultaneously to all destinations, then interarrival times t_s and t_p will be identical.

- If transmission delays of $seq(m_1)$ and $seq(m_2)$ from s to p are the same, then p can use the value of t_{sp} to locally determine t_s. This avoids assumptions on the drift rate of clocks.

Process p can easily calculate the delay it should have introduced to the optimistic delivery of m_2 to match its relative distance from p_1 and p_2 to that of the sequencer. Specifically, it should have delayed m_2 by $t_{sp} - t_p$.[2] To cope with spurious variations on transmission delays, adjustments are made taking into account an inertia pondering factor.

The next section presents a detailed description of how the delays are calculated and which messages are delayed. Right now, it is worth note that delays to a process messages are only introduced when it is not possible to achieve the same result by reducing the delays inflicted to the other.

The way the sequencer calculates its own messages delays is different. Should it use the same method as the others and it, clearly, would not delay its own messages. To understand the method followed by the sequencer lets first exemplify the consequences of not introducing delays on the sequencer's own messages. Consider three processes, p, q and s. Process s is the sequencer. Process q, for simplicity, is δ equidistant of s and p. The distance from s to p is d_{sp} and the distance of s to itself is d_{ss}.

Having p and s relatively equidistant from q means that $(\delta - d_{ss}) = (\delta - d_{sp})$. To achieve this, since d_{sp} cannot be reduced, the possibilities are 1) p to distance from q, or 2) s to distance from itself, or both. Now suppose that s does not delay its own messages. In this case, p will have to stand back $\Delta = d_{sp} - d_{ss}$ from q. Since d_{ss} (the loopback delay) is usually negligible it can be assumed that $\Delta \simeq d_{sp}$. This means that when a message multicast by q is optimistically delivered at p it is almost simultaneously delivered authoritatively at p too. Therefore, unless the sequencer delays the optimistic delivery of its own messages the size of the optimistic window at the other processes becomes uninteresting or even vanishes.

In the next section, we show how the sequencer computes the delay for its own messages. This, contrary to other processes adjustments, is not independent and requires their cooperation. The idea is that the sequencer will stand back from itself as it distances from its farthest process.

[2] Notice that t_p is negative in Figure 4.3, indicating that the relative order of m_1, m_2 is reversed.

4.2.3 Algorithm

In Figure 4.4 we present the statistically estimated optimistic total order algorithm as executed by each process. The algorithm consists of a procedure *TO-multicast(m)* invoked by the client application to multicast a message and a set of four *upon-do* statements, executed atomically, that deal with the optimistic and authoritative delivery of the messages. The actual delivery of the messages to the client application is done through two upcalls *opt-deliver(m)* and *fnl-deliver(m)*. Procedure *adjust* is an auxiliary procedure local to the algorithm.

Each process manages four queues R, O, F and S where it keeps track of the messages received, optimistically and authoritatively delivered to the application, and those for which it has already received a sequence number, respectively. Every message *m* has a special attribute (*m.sender*) identifying its sender. At each process a variable *seq* identifies the sequencer process.

To multicast a totally ordered message, the client application invokes procedure *TO-multicast(m)* (lines 9-11). This, in turn, invokes an underlying primitive providing reliable multicast with a pair $(m, max(delay[]) - delay[seq])$. The value computed by $max(delay[]) - delay[seq]$, as discussed below, corresponds to the delay the process suggests the sequencer to inflict to its own messages.

The reception and delivery of messages is done by the four *upon-do* statements. The first two handle the optimistic delivery while the others handle the authoritative delivery.

When a process p receives a message m (line 12) it simply adds m as a tuple (m, d, d') to the queue, R, of received messages scheduling its delivery for after the delay d' inflicted by p to the sender of m. When this timer expires and if m was not already optimistically delivered ($m \notin O$) nor authoritatively delivered ($m \notin F$), which corresponds to the condition on line 15, then m is optimistically delivered to the application and the fact registered by adding m to the O queue. If p happens to be the sequencer it computes a sequence number to give to m and reliably multicasts a sequence message composed by m's id and its sequence number. Afterwards, p (if in the role of sequencer) takes parameter d just received with m and adjusts the delays it imposes to its own messages.

Upon receiving a sequence message at line 25, each process simply adds the received tuple (message id and sequence number) plus the current time to the queue of sequence numbers S.

Once a message m that has already been received ($m \in R$) gets a sequence number in the S queue and its sequence number corresponds to the next message to be authoritatively delivered (the whole condition at line 28), then m is authoritatively delivered to the application through *fnl_deliver*. At this point, the algorithm

```
$g \leftarrow 0$ {Global sequence number}
$l \leftarrow 0$ {Local sequence number}
$R \leftarrow \emptyset$ {Messages received}
$S \leftarrow \emptyset$ {Sequence numbers}
$O \leftarrow \emptyset$ {Messages opt-delivered}
$F \leftarrow \emptyset$ {Messages fnl-delivered}
$delay[1..n] \leftarrow 0$
$r\_delay[1..n] \leftarrow 0$ {Delays requested to the sequencer}

procedure $TO\_multicast(m)$ do
    $R\_multicast(\text{DATA}(m, max(delay[]) - delay[seq]))$
end procedure

upon $R\_deliver(\text{DATA}(m, d))$ do
    $R \leftarrow R \cup \{(m, d, now + delay[m.sender])\}$
end upon

upon $\exists(m, d, t) \in R : now \geq t \wedge m \notin O \wedge m \notin F$ do
    $opt\_delivery(m)$
    $O \leftarrow O \cup \{m\}$
    if $p = seq$ then
        $g \leftarrow g + 1$
        $R\_multicast(\text{SEQ}(m, g))$
        $r\_delay[m.sender] \leftarrow d$
        $delay[p] \leftarrow max(r\_delay[])$
    end if
end upon

upon $R\_deliver(\text{SEQ}(m, s))$ do
    $S \leftarrow S \cup \{(m, s, now)\}$
end upon

upon $\exists(m, d, o) \in R : (m, l + 1, t) \in S \wedge m \notin F$ do
    fnl_delivery(m)
    if $\exists(m', d', o') \in R : (m', l, t') \in S$ then
        $\Delta \leftarrow (t - t') - (o - o')$
        if $\Delta > 0$ then
            $adjust(m'.sender, m.sender, \Delta)$
        else
            $adjust(m.sender, m'.sender, |\Delta|)$
        end if
    end if
    $l \leftarrow l + 1$
    $F \leftarrow F \cup \{m\}$
end upon

procedure $adjust(i, j, d)$ do
    $v \leftarrow (delay[i] \times \alpha) + (delay[i] - d) \times (1 - \alpha)$
    if $v \geq 0$ then
        $delay[i] \leftarrow v$
    else
        $delay[i] \leftarrow 0$
        $delay[j] \leftarrow delay[j] + |v|$
    end if

end procedure
```

Figure 4.4: Delay compensation algorithm for process p

computes the adjustments that might need to be done to the delays inflicted to the sender of m or to the sender of the message m' delivered just before m. To do this it considers the interval between the reception of the sequence number for m' and the sequence number for m given by ($t - t'$) and the interval between the optimistic reception of m' and the optimistic reception of m given by ($o - o'$). The difference Δ (line 31) between these intervals represents the relative adjustment that should have been done to the delays imposed to the optimistic delivery of m' or m to make the interval of the optimistic deliveries of these messages match that of the authoritative deliveries.

If Δ is negative it means that the optimistic order matched the authoritative order. If Δ is positive then either the order was reversed or the interval between optimistic deliveries is smaller than the interval between authoritative deliveries. Depending on this, procedure $adjust$ is called differently. In the first case *adjust* is called to decrease the delay put on messages received from the sender of m, otherwise it should decrease the delay inflicted to the sender of m'.

Procedure *adjust(p,q,d)* works as follows. Based on the delay d and on a inertia parameter α, it computes the new delay v to give to messages of p. If v becomes negative, this means that it actually needs to anticipate p's messages which is not possible. Instead, it does not delay the messages of p but starts delaying the messages of q by an additional $|v|$ amount.

Finally, the sequencer computes the delays on the optimistic delivery of its own messages as follows. Every process, when $R\_multicasts$ a data message (line 10), also sends the value of the greatest delay it is applying locally (this is usually the self delay) minus the delay it is currently applying to the sequencer messages. Only the sequencer makes use of this values keeping track of them on vector r_delay. The delay the sequencer inflicts on its own messages is given, at each moment, by the greatest value in r_delay.

4.3 Total Order Protocols Implementation

This section describes the implementation of the statistically estimated optimistic total order protocol, in a Java based group communication toolkit [PO05]. We describe the architecture of the group communication toolkit and then the protocol implementation.

4.3.1 Architecture and Interfaces

The implementation of the communication protocols in the group communication toolkit uses a layered architecture depicted in Figure 4.5. Upon initialization, the group communication toolkit offers an implementation of the `Top` interface, and an application using group communication must implement the `Bottom` interface, depicted in Figure 4.6 along with the JAVA classes used in the implementation of the total order protocols. A communication protocol layer is a piece of software implementing both the `Top` and `Bottom` interfaces and ensuring certain properties on the messages it delivers to upper layers.

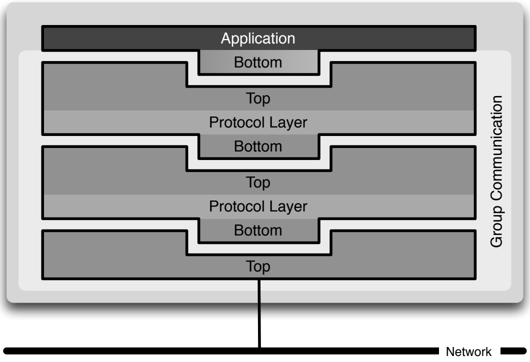


Figure 4.5: Protocols implementation architecture.

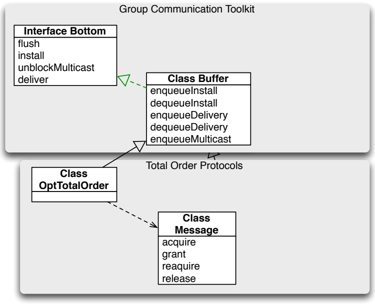


Figure 4.6: Implementation classes and interfaces.

In order to facilitate the development of new communication protocol layers the group communication toolkit defines a `Buffer` class implementing both the `Top` and `Bottom` interfaces.

The Top interface defines methods for initialization, init, for sending messages to the group, multicast, for protocol composition, i.e. to stack communication protocol layers, stack, for upper layers to signal they are ready to view change, done, and to signal they are ready to deliver messages, unblockDeliver.

The Bottom interface defines the counterpart methods of the Top interface which are the flush called by the lower layer before view changing, in order to the layer ensure that all messages that must be delivered in the current view are effectively delivered. When all messages are delivered then the done method on the lower layer is called to signal that the layer is ready for the view change. The install method is called to view change, i.e., to change the group membership, by setting the current group members. The unblockMulticast is called by the lower layer to signal that it can resume sending messages. The lower layer signals that message exchanging is blocked by returning false after multicasting a message. Finally, the deliver method is used to receive messages, multicasted by a group member.

The Buffer class as stated earlier implements both the Top and Bottom interfaces and defines the methods to abstract the communication protocol layer as a queue. Using this base class, a communication protocol layer may intercept messages entering the queue (methods with the enqueue prefix) or before leaving the queue (methods with the dequeue prefix), in order to enforce the desired message properties. For instance, a total order protocol may intercept unordered messages when entering the queue in order to order them, and it may also delay their delivery intercepting the dequeueDelivery and retaining the messages until the desired properties are achieved.

4.3.2 Total Order Protocol Implementation

The logic behind the implementation of the sequencer based total order protocol is depicted in Figure 4.7. The protocol manages two queues, the rcvOPT queue holding messages broadcasted for which the order is not yet established, and the rcvFNL holding the sequence numbers for messages in the rcvOPT queue.

The implementation intercepts messages entering the protocol layer, by redefining method enqueueDelivery, depicted in Listing 4.1. This step is only meaningful for the sequencer and if it is not an ordered message. In the case of the process intercepting the message being the sequencer (line 2), the message is decoded (line 3) and if it is an ordinary message, i.e., not an ordered message, the sequencer's assign it a sequence number by calling the order procedure (line 5). Afterwards the message is reassembled (line 6) and inserted in the Buffer queue (line 8).

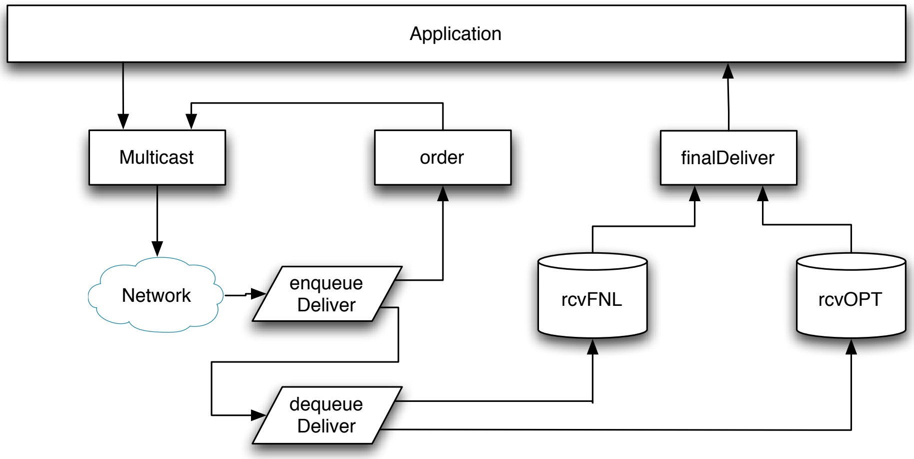


Figure 4.7: Total order protocol implementation.

```
1   void enqueueDelivery(int i, Message obj) {
      if(sequencer){
        FastABHeaders header=FastABHeaders.pop(obj);
        if(!header.ordered())
5             order(new FastABHeaders(header));
        header.push(obj);
      }
      super.enqueueDelivery(i,obj);
    }
10
    void order(FastABHeaders nh){
      nh.set_ord(vid,last++);
      Message m=new Message();
      nh.push(m);
15    enqueueMulticast(m);
    }
```

Listing 4.1: Message ordering.

The `order` procedure assigns to the message the next sequence number in the current view (line 12), creates a new message and multicasts it (lines 13-15).

The implementation also intercepts messages before being delivered for the upper layer in order to ensure they are only delivered when their sequence number is known and all previous messages have been delivered. This is done by redefining the `dequeueDelivery` method depicted in Listing 4.2. It starts decoding the message (line 2) and depending on being an ordered or ordinary message it is inserted in the `rcvFNL` or `rcvOPT` respectively (lines 4 or 6). Afterwards a verification is done to check whether the received message fulfills the conditions to be delivered in total order to the upper layer, by calling the `checkordered`

```
1  void dequeueDelivery(int i, Message obj) {
     ABHeaders header=ABHeaders.pop(obj);
     if(header.ordered())
       rcvFNL.put(new Integer(header.get_ord()), header);
5    else
       rcvOPT.put(header.message_id(),new buf_message(i,obj));
     checkordered();
   }

10 void checkordered(){
     ABHeaders header=(ABHeaders)rcvFNL.get(new Integer(last_dlvr+1));
     if(header!=null){
       buf_message m=(buf_message)rcvOPT.remove(header.message_id());
       if(m!=null){
15       rcvFNL.remove(new Integer(++last_dlvr));
         super.dequeueDelivery(m.i,m.m);
         check_ordered();
       }
     }
20 }
```

Listing 4.2: Message delivery.

procedure (line 7).

The `checkordered` procedure starts by checking if there is a message identifier associated to the next sequence number to be delivered to the application (lines 11-12). If it is the case, then it checks if it have already received the message associated to the message identifier (lines 13-14). If the previous check is successful, then the sequence number of the next message to deliver is removed from the `rcvFNL` queue and the sequence number for the next message to be delivered updated (line 15). The message is then queued for delivery to the upper layer (line 16). Afterwards the procedure is recalled in order to verify if it is possible to deliver another message.

4.3.3 Statistically Estimated Optimistic Total Order Protocol Implementation

The implementation of the statistically estimated optimistic total order protocol requires several changes from the total order implementation of the previous section. These changes are depicted in Figure 4.8 and are related to the ordering procedure, the processing of optimistic messages and the delivery of totally ordered messages to the upper layer.

In addition to the two queues used by the total order protocol, the statistically estimated optimistic total order protocol defines the `optDLVR` and `optDELAY` queues. The first one stores messages optimistically delivered to the upper layer

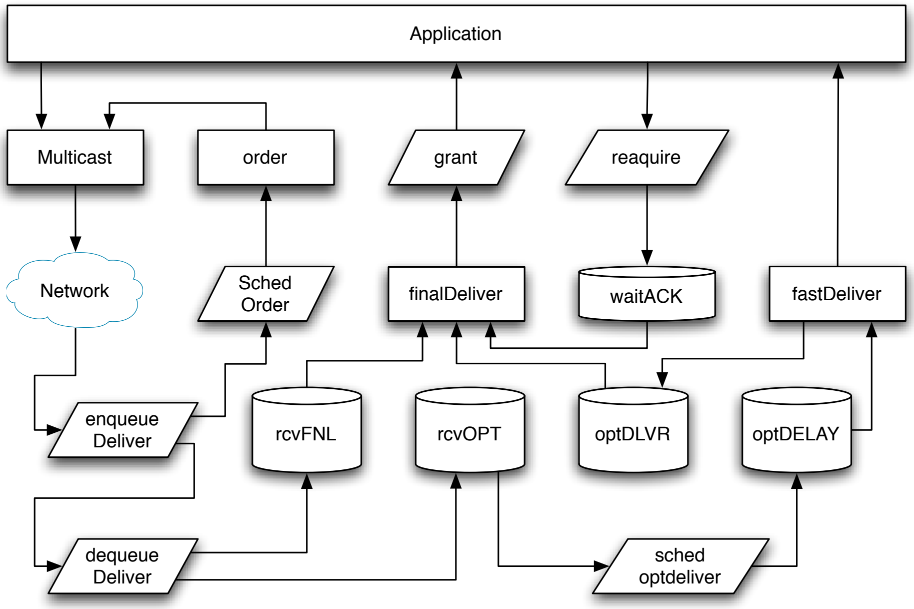


Figure 4.8: Statistically estimated optimistic total order protocol implementation.

while the second stores messages waiting to be optimistically delivered to the upper layer.

The statistically estimated optimistic total order protocol, requires messages to be delivered twice to the upper layer. This is unfortunate as it requires changing the `Bottom` interface, thus requiring changes in the protocol layer interface. In order to avoid changing the protocol layer interface, the adopted solution only delivers the message once, and then notifies the upper layer informing it that some or all of its desired properties are met.

The solution uses the `Message` class of Figure 4.6. In it, the `acquire` is used by the upper layer to query the message for checking whether the required property is already met, or otherwise to set a callback for being notified when such condition is met. The communication protocol layer sets properties to the message by calling the `grant` method, and may want to be notified when such property is known by the upper layer, by calling the `reacquire` method. After knowing that the required message property is met, the upper layer should also notify the communication protocol layer, by calling the `release` method.

This mechanism is used by the statistically estimated total order protocol in order to ensure that the total order property of the next ordered message is only set when the previous one has been processed by the upper layer. The protocol uses the `waitACK` as a mutex to control when the total order property of the next ordered message can be set.

```
 1  void enqueueDelivery(int i, Message obj) {
      if(sequencer){
        FastABHeaders header=FastABHeaders.pop(obj);
        if(!header.ordered())
 5        if(i==0 && delay[0]!=0)
            order_coord(i,new FastABHeaders(header));
          else
            order(i,new FastABHeaders(header));
        header.push(obj);
10    }
      super.enqueueDelivery(i,obj);
    }

    void order_coord(final int i,final FastABHeaders header){
15    SeqProcess.self().schedule( new Runnable() { public void run() {order(i,header);} }, delay[0]/1000);
    }

    void order(int i,FastABHeaders nh){
      order(nh);
20    if(i!=0){
        dincr[i]=nh.get_delay();
        delay[0]=max(dincr);
      }
    }
```

Listing 4.3: Message ordering.

In order to set sequence numbers to messages, the statistically estimated total order protocol also redefines the `enqueueDelivery` method. As depicted in Listing 4.3, the changes relatively to the total order protocol are when the sequencer wants to set sequence numbers to messages issued by the himself (line 5). In this case, instead of calling immediately the order procedure, it calls the `order_coord` which schedules the call to the order procedure to after the delay the sequencer imposes to its own messages elapses. The `order` procedure (lines 18-24), uses the `order` procedure of the sequencer based total order protocol and in the case of messages not issued by the sequencer himself (line 20), the sequencer updates the maximum delay the sender is imposing on messages before optimistically delivering them (line 21). Afterwards the sequencer sets the delay it imposes on its own messages to be equal to the highest delay some process is imposing locally (line 22).

Messages are intercepted before being delivered to the upper layer by redefining the `dequeueDelivery` method depicted in Listing 4.4. It starts by decoding the message (line 2) and depending on being an ordered or ordinary message it is placed in the `rcvFNL` or `rcvOPT` queue. Afterwards the `processmessages` procedure which identifies the messages that may be processed is called and acts accordingly. If there are ordered messages and the protocol is not waiting from a notification from the upper layer (line 11) then it calls the procedure responsible for the processing of ordered messages which is depicted in Listing 4.6. if there are ordinary messages received and not yet processed (line 13) the the procedure responsible for the processing of ordinary messages, depicted in Listing 4.5 is called.

```
1   void dequeueDelivery(int i, Message obj) {
      FastABHeaders header=FastABHeaders.pop(obj);
      if(header.ordered())
        rcvFNL.addLast(new buf_message(header,obj));
5     else
        rcvOPT.addLast(new buf_message(header,obj));
      processmessages();
    }

10  void processmessages(){
      if(rcvFNL.size()!=0 && ! waitACK)
        processrcvFNL();
      if(rcvOPT.size()!=0)
        processrcvOPT();
15  }
```

Listing 4.4: Message delivery.

```
1   void processrcvOPT(){
      m=(buf_message)rcvOPT.removeFirst();
      if(delay[m.hd.sender()]!=0){
        optDELAY.addLast(m);
5       sched_fastdelivery(m.hd.sender(),m.m,m.hd);
      }else
        fast_delivery(m.hd.sender(),m.m,m.hd);
    }

10  void fast_delivery(int i, Message m, FastABHeaders header){
      optDLVR.addLast(new buf_message(header,m,System.currentTimeMillis()/1000));
      super.dequeueDelivery(i,m);
      processmessages();
    }
15
    void sched_fastdelivery(final int i,final Message m, final FastABHeaders header){
      SeqProcess.self().schedule( new Runnable() {
      public void run() {
        if(check_queue(new buf_message(header,m),optDELAY) != null)
20        fast_delivery(i,m,header);
      } },delay[i]/1000);
    }
```

Listing 4.5: Processing optimistic messages.

The processing of ordinary messages depicted in Listing 4.5 removes the first message from the `rcvOPT` queue (line 2), and in the case that its delivery should be delayed (line 3), it is placed in the optDELAY queue and the procedure to schedule (sched_fastdelivery) its delivery to after the delay it should suffer elapses called (line 5). Otherwise the procedure to deliver it to the upper layer (fast_delivery) is called (line 7).

The sched_fastdelivery schedules the execution of the run procedure depicted in lines 19-20 to be run after the delay messages from sender i should suffer elapses. The procedure checks if the message is still in the optDELAY queue (line 19) in which case it is removed from the queue and the fast_delivery procedure called in order to optimistically deliver the message to the upper layer. This check is necessary as the processing of totally ordered messages may force the message to be delivered before the delay elapses.

The fast_delivery procedure (lines 10-14) places the message and the instant of its optimistic delivery in the optDLVR queue (line 11), queues it for delivery to the upper layer (line 12) and finally recalls the processmessages procedure of Listing 4.4, to check if it is possible to process some other message (line 13).

The procedure responsible for the processing of ordered messages, depicted in Listing 4.6 is divided in three steps. In the first, lines 3-4 checks whether it is possible to set the total order property on the next message to be delivered in total order. The second, lines 5-12, checks if the message has not yet been delivered because of the delays imposed to it. The third step, lines 14-18, delivers the message in total order.

```
 1  void processrcvFNL(){
      buf_message m;
      if( waitACK || (m=get_ord_msg(last_dlvr+1))== null)
        return;
 5    buf_message m1=check_queue(m,this.optDLVR);
      if(m1==null){
        rcvFNL.addFirst(m);
        m1=check_queue(m,this.optDELAY);
        if(m1==null)
10        m1=check_queue(m,this.rcvOPT);
        if(m1!=null)
          fast_delivery(m1.hd.sender(),m1.m,m1.hd);
      }else{
        m1.m.grant(FINAL);
15      if(!m1.m.reaquire(FINAL,cb))
          waitACK=true;
        last_dlvr++;
        adjust_delays(m1);
        processmessages();
20    }
    }
```

Listing 4.6: Processing ordered messages.

The first phase of the processing of ordered (lines 3-4), is a validation phase. Checks are made in order to establish whether it is possible to deliver the next ordered message and if there is an ordered message identifying the next message to be delivered in total order to the upper layer. If any of these conditions fails then the processing stops (line 4). Afterwards, the `optDLVR` queue is checked to verify whether the message has been optimistically delivered or not. If the message is not in the `optDLVR` queue, i.e., it has not been optimistically delivered, then it may be in the `optDELAY`, waiting to be delivered, or in the `rcvOPT` queue if it has just been received, or it may not yet been received. In any case, the ordered message is placed in the head of the `rcvFNL` queue (line 7), because the optimistic message has not yet been delivered to the upper layer, and the total order property can not be granted. If the message is in one of the `optDELAY` or `rcvOPT` queues it is removed from there (lines 8-10) and is optimistically delivered (line 12), by the `fast_delivery` procedure of Listing 4.5.

If the message has already been optimistically delivered to the upper layer, then the total order property is granted (line 14), and a notification callback is set in the case that the upper layer do not immediately acknowledges the grant operation, in which case the `waitACK` is set to `true` (lines 15-16). Afterwards, the order number for the next message to be delivered is set (line 17), the procedure to adjust the delays to impose to the optimistic delivery of messages called (line 18), and the `processmessages` procedure in order to verify whether it is possible to deliver some other messages (line 19).

```
void adjust_delays(buf_message m){
  if(lastfinal!=−1 && !sequencer){
    long delta=(long)((System.currentTimeMillis()/1000−lastfnal)−(m.time−lastfast));
    if(delta>0)
      adjust(lastsender,m.hd.sender(),delta);
    else if(delta<0)
      adjust(m.hd.sender(),lastsender,−delta);
  }
  lastfast=m.time;
  lastfnal=System.currentTimeMillis()/1000;
  lastsender=m.hd.sender();
}

void adjust (int p, int q, long d){
  fut=(long)(delay[p]∗alfa + (delay[p]−d)∗(1−alfa));
  if(fut<0){
    delay[p]=0;
    delay[q]=delay[q]−fut;
  }else
    delay[p]=fut;
}
```

Listing 4.7: Adjustments to message delays.

The adjust_delays procedure depicted in Listing 4.7 starts by verifying that this is not the first total order delivery and that the process delivering the message is not the sequencer (2). If this is the case, then it calculates Δ, i.e., the difference between the intervals of the total order deliveries and of the optimistic deliveries of this and of the previously totally ordered message (line 3). Depending on the value of Δ the adjust procedure is called to reduce the delay inflicted on messages sent by the sender of the previously total order delivered message, if $\Delta > 0$ (line 5), or to reduce the delay inflicted to messages whose sender is the same as the one sending the message being delivered (line 7). Finally the time of the last optimistic and final deliveries is updated as well as the indication of the sender of the last total order delivered message.

The adjust procedure, starts by calculating the delay that should be set on messages of sender p (line 15). If this value turn out to be negative, this means that messages from sender p shothe uld optimistically delivered before being received. As this is impossible to happen the solution is to stop delaying messages from sender p (line 17) and enlarge the delay imposed on messages from server q (line 18). If the calculated value is positive, then it is the delay applied to messages from server p before optimistically deliver them.

4.4 Protocol Evaluation

This section describes the evaluation of the statistically estimated total order protocol comparing it to the sequencer based total order protocol.

The evaluation starts with a simulation of both protocols recording for each process the instants the messages are sent, received, optimistically and authoritatively delivered. After the simulation runs the recorded values are used to calculate protocol spontaneous total order, optimistic and delivery windows, used to demonstrate that the statistically estimated total order protocol only marginally increases the protocol latency, and only because of the delays the sequencer imposes to self messages.

The values obtained by the simulation were afterwards confirmed by a prototype based evaluation of the protocols.

4.4.1 Evaluation Criteria

For an application to benefit from optimistic ordering it is required that *i*) optimistic order closely matches the final order; and *ii*) the time between optimistic delivery and final delivery is enough to do meaningful processing. Performance evaluation is done with an event-based simulation, which allows to study the impact of the system and the protocol parameters, as well as with an implementation of the protocol within a group communication toolkit, allowing to confirm the simulation results.

The primary evaluation criteria is thus to compare optimistic and final orders of both the sequencer based total order multicast protocol and the statistically estimated optimistic total order multicast protocol. To accomplish this, for an execution of each protocol, the sequence of optimistic and final deliveries are logged. After the execution has finished, the first final delivery and the first optimistic delivery are compared. If they match a hit is recorded, otherwise a miss is recorded. The messages are then removed from both logs and the process repeated until the logs are empty. From these, the ratio of correctly ordered optimistic messages is derived. This closely matches the impact on the application: If an optimistic delivery is much earlier than it should, it will cause several misses until it is removed from the log.

A second evaluation criterion is to compute whether the time between optimistic delivery and final delivery — the optimistic window — is enough to do meaningful processing. Finally, the impact of delay compensation in end-to-end latency of the final delivery is also studied. It has to be ensured that the improvements in spontaneous total order are not obtained at the expense of larger end-to-end latency. It is, however, expectable some increase in end-to-end latencies due to the delay introduced by the sequencer to its messages. This delays the ordering of sequencer's messages, thus increasing end-to-end latency, both in the sequencer and in the other processes.

4.4.2 Simulation Based Evaluation

Using discrete event simulation the performance of the protocol is studied in a scenario without application or message processing and delivery overheads. We consider a fully connected point-to-point network. The transmission delay in each link is normally distributed with parameterized mean and standard deviation. Message inter-arrival rate is exponentially distributed with equal mean in every participant.

In the following the several network topologies used in the evaluation of the protocol are described. All settings have five processes, a reference latency of 30 ms, and use process 1 as the sequencer.

Network topology 1 In this topology the latency between every two different processes is 30 ms.

Network topology 2 In this topology the latency between processes i and j ($1 \leq i, j \leq 5$ and $i \neq j$) is calculated as $(7 - j) * (7 - i) * latency$ which results in the latencies described in Table 4.1.

Processes	1	2	3	4	5
1	0	900	720	540	360
2	900	0	600	450	300
3	720	600	0	360	240
4	540	450	360	0	180
5	360	300	240	180	0

Table 4.1: Latencies between processes for topology 2, in ms.

Network topology 3 This topology mimics a situation where a long distance link separates two clusters. In this setting, processes 1 and 2 form cluster 1, and processes 3, 4 and 5 form cluster 2. Latencies between processes inside each cluster is 30 ms while the latency between processes not in the same cluster is 60 ms.

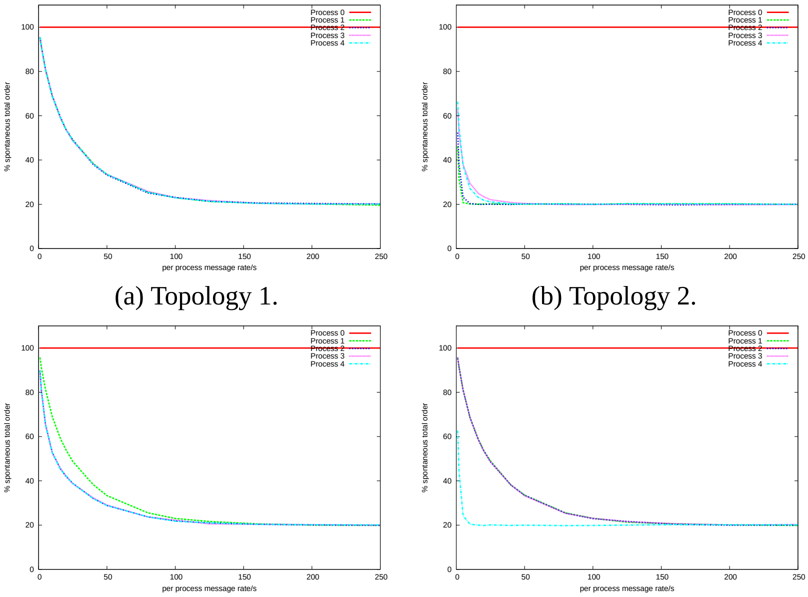
Network topology 4 This topology is similar to the previous one but now 4 processes are in one cluster with a latency of 30 ms, while the remaining is isolated having a latency of 300 ms.

The experiences conducted evaluate the sequencer based total order protocol and ours under different loads. In each run, the per process message rate ranges from 1 to 200 messages per second, and the standard deviation of message transmission delays is of 0 and 10%.

Figure 4.9, presents the spontaneous total order of the sequencer based total order protocols as observed by each process and for each one of the topologies considered in the experiences. The values presented consider a standard deviation of message transmission delay of 0%.

While, by construction, the sequencer as a 100% hit ratio, it can be observed that spontaneous total order at the other processes has a quick degradation.

The point-to-point latencies and message rate also influence the spontaneous total order. In a network with equal point-to-point latencies, all processes, except the sequencer, observe similar spontaneous total order (Figure 4.9a) which decreases



(a) Topology 1.

(b) Topology 2.

(c) Topology 3.

(d) Topology 4.

Figure 4.9: Sequencer based protocol spontaneous total order ($\sigma = 0\%$).

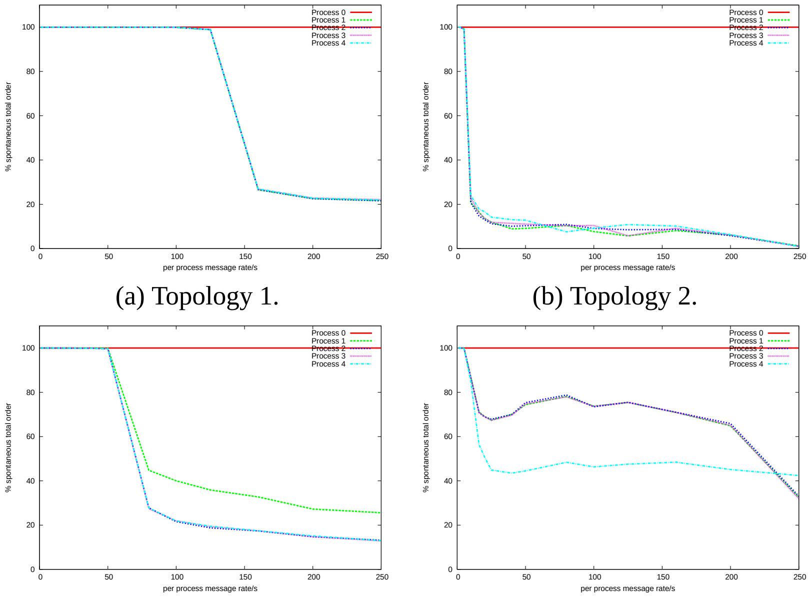
with the load in the system. The differences in the latencies of processes in different clusters of topologies 3 and 4 is also observable in the spontaneous total order by the distance among the lines of Figures 4.9c and 4.9d.

The sequencer based total order protocol is not very sensitive to message transmission delays. The use of a standard deviation of message transmission delays of 10% only marginally affects the spontaneous total order observed.

The spontaneous total order for the statistically estimated optimistic total order protocol, depicted in Figure 4.10, presents better results except for topology 2. The differences in the point-to-point latencies are much more noticeable in the optimistic total order than in the total order protocol. Despite having better results in the spontaneous total order, the processes of each cluster are easily identified from Figures 4.10c and 4.10d than they were in Figures 4.9c and 4.9d

The statistically estimated optimistic total order, is much more sensible to variations of message transmission delays. Even so, for loads under 50 messages per second it usually presents better spontaneous total order than the sequencer based total order protocol.

In Figure 4.11 the improvements in spontaneous total order resulting from the statistically estimated optimistic total order multicast protocol are presented. The



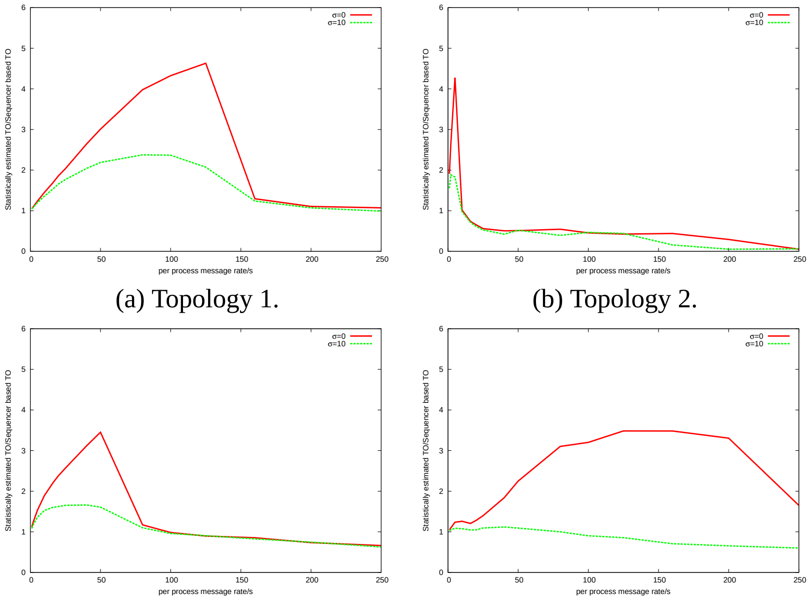
(a) Topology 1.

(b) Topology 2.

(c) Topology 3.

(d) Topology 4.

Figure 4.10: Statistically estimated protocol spontaneous total order ($\sigma = 0\%$).



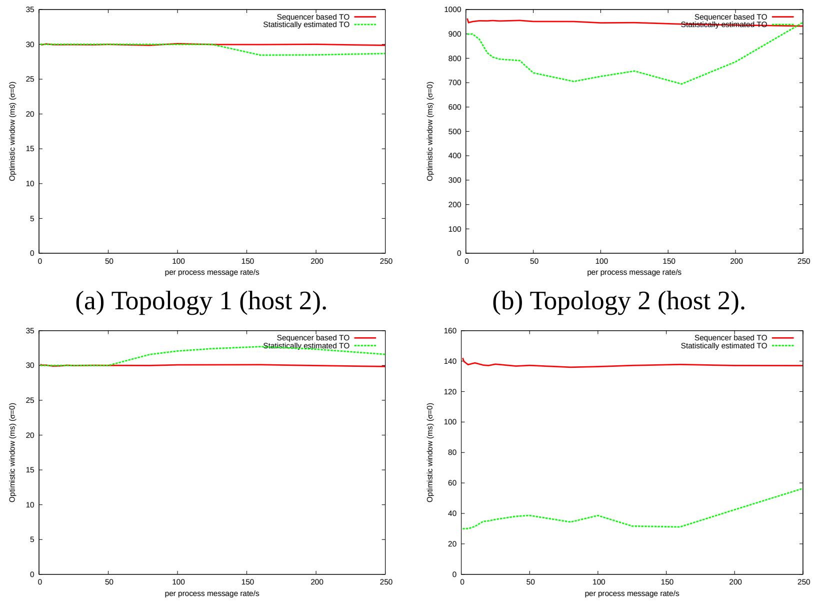
(a) Topology 1.

(b) Topology 2.

(c) Topology 3.

(d) Topology 4.

Figure 4.11: Improvement in spontaneous total order.



(a) Topology 1 (host 2).

(b) Topology 2 (host 2).

(c) Topology 3 (host 2).

(d) Topology 4 (host 4).

Figure 4.12: Optimistic windows.

improvements depend on the network topology, variation of message transmission delays and message rates. Higher variations on message transmission delays imply lower improvements on the spontaneous total order.

For message transmission delays with standard deviation of 10%, the advantages of the statistically estimated optimistic total order multicast protocol are only marginal, but having such a variation in message transmission delays it is not a very common situation. As observed in extensive measurements of the Internet [Pax97], the standard deviation of transmission delays is mostly less than 10% for large data packets and often less than 1% for small control packets. From this, it is expectable that the statistically estimated optimistic total order multicast protocol present better values when ordering small messages. In the case of large messages it is also expected that messages can be split, one small message used by the ordering protocol and large ones carrying the bulk of the data.

With respect to spontaneous total order, the usage of the optimistic protocol is advantageous. It remains to be established what happens with respect to the optimistic window and protocol latency.

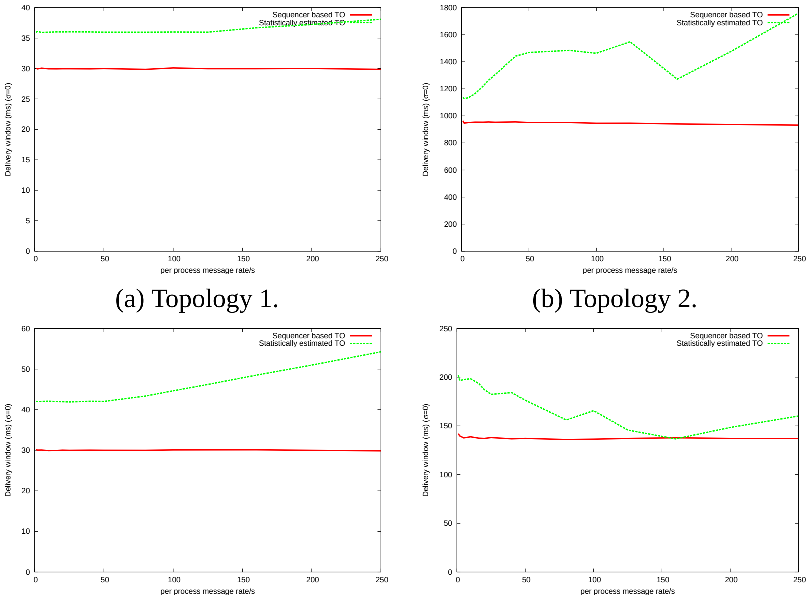
Figure 4.12 depicts for each of the studied network topologies the size of the optimistic window.

For network topology 1, the size of the optimistic window is $30\ ms$ and is almost independent from the per process message rate.

For network topology 2 each process presents different optimistic windows, and the optimistic protocol clearly reduces the size of this window. Nevertheless the size of this optimistic window is always higher than $600\ ms$.

Network topologies 3 and 4 present two distinct values for the optimistic window. One for the processes in the group of the sequencer, and another for the processes in the other group.

The optimistic window measures the interval of elapsed time between the instant a message is optimistically delivered and the instant its order is established. An important aspect of the statistically estimated total order protocol is that it should not delay the delivery of the authoritatively ordered message. In order to establish that the statistically estimated total order do not increases the protocol latency, we consider delivery window which measures the interval between the message reception and its authoritative delivery. This interval should be the same for both protocols and is equal to the optimistic window for the sequencer based total order protocol, which optimistically delivers the message immediately after its reception.



(a) Topology 1.

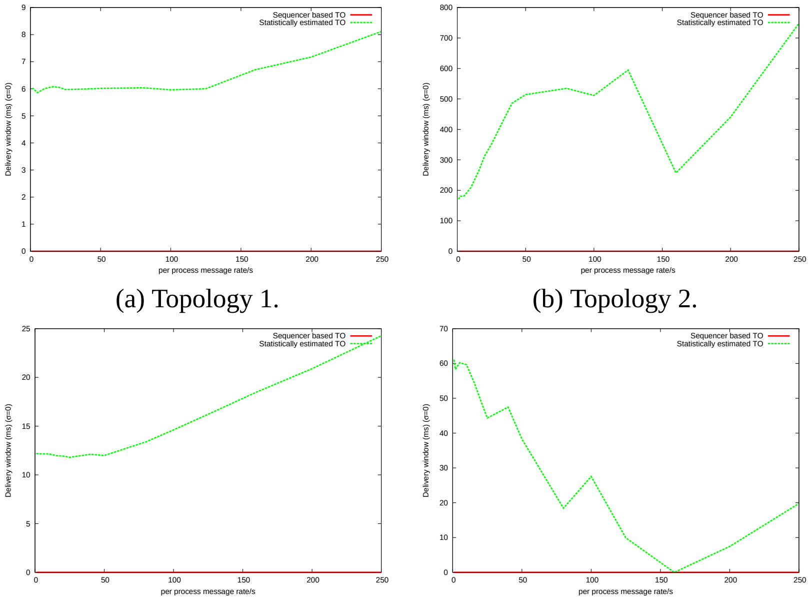
(b) Topology 2.

(c) Topology 3.

(d) Topology 4 (process 4).

Figure 4.13: Delivery windows.

The values of delivery window are presented in Figure 4.13. As expected, they are equal to the optimistic window for the sequencer based total order protocol. The



(a) Topology 1.

(b) Topology 2.

(c) Topology 3.

(d) Topology 4.

Figure 4.14: Sequencer delivery window.

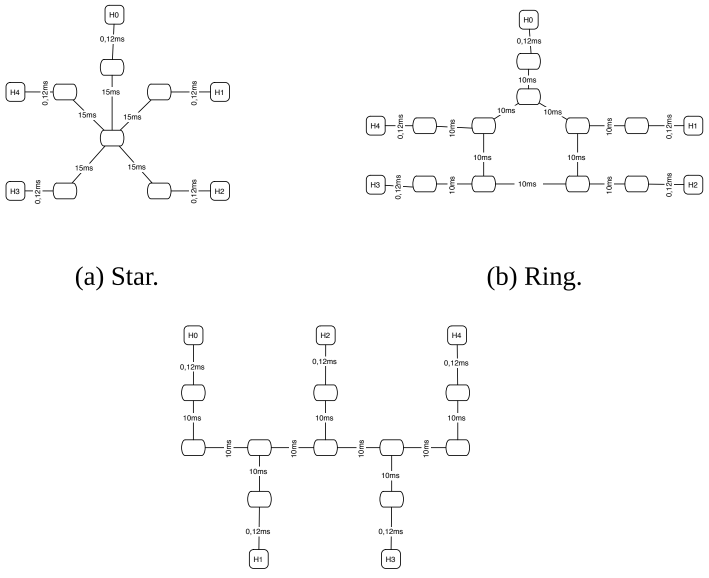
statistically estimated total order protocol presents higher delivery window. The higher delivery window of the statistically estimated total order protocol results from the fact that messages from the sequencer are not ordered immediately after being received but after suffering the delay the sequencer is imposing to its own messages. This results in the sequencer having a delivery window different of 0 which is reflected in the other processes delivery window.

The sequencer's delivery window is presented in Figure 4.14. It can be observed, from Figures 4.13 and 4.14 that the differences in delivery windows are equal to the sequencer delivery window.

4.4.3 Prototype Based Evaluation

In order to validate the results obtained with simulation, the performance of the sequencer based and statistically estimated optimistic total order protocols implementation of Section 4.3 have been evaluated.

Measurements presented below were obtained by running the protocol implementations on top of the simulation infrastructure described in Section 5.4.2.



(a) Star.

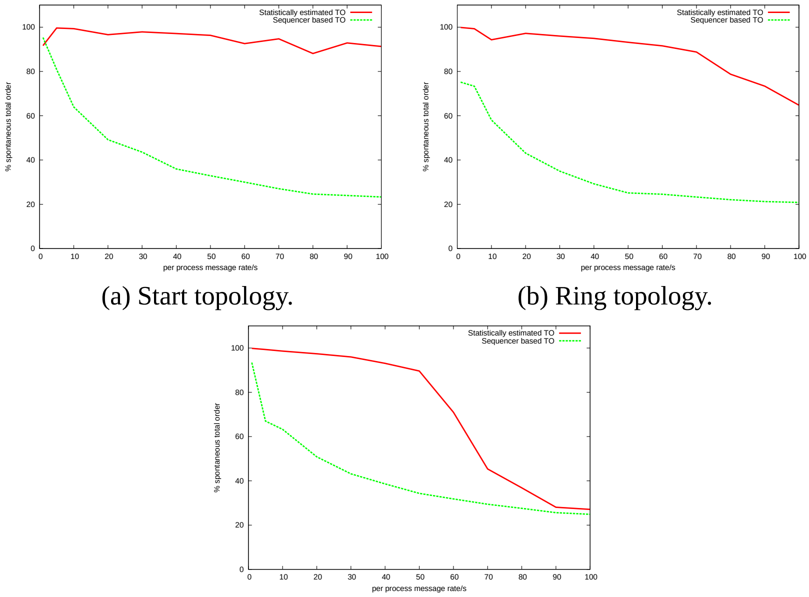
(b) Ring.

(c) Bus.

Figure 4.15: Network topologies.

The protocols were evaluated in the 3 network topologies depicted in Figure 4.15. All the experiences were conducted with 5 processes sending messages at a similar data rate. The star topology (Figure 4.15a) is a symmetric network composed of a central router to which all the leaf networks connect through a 15 ms WAN link. In the backbone network of the ring topology (Figure 4.15b), each router is connected to two adjacent routers, through a 10 ms WAN link, forming a ring. Each of the leaf networks is connected to one of those routers through a 10 ms WAN link. In the bus topology (Figure 4.15c), the backbone network is composed of 3 inner routers and 2 outer routers. Each outer router connects to one of the inner routers, and the inner routers connect to two routers of the backbone network. Each connection as a 10 ms latency. As in the ring topology, each of the leaf networks is connected to one of the backbone routers through a 10 ms WAN link. Common to all topologies is the fact that adjacent processes are separated by a latency of 30 ms.

In the experiences, message inter-arrival for each process is obtained from a negative exponential distribution with the corresponding parameter to accomplish the desired message rate. Each process records the instant it sends and receives each message optimistically and authoritatively. After finishing the experience, those records are processed and the value of the spontaneous total order calculated. The



(a) Start topology.

(b) Ring topology.

(c) Bus topology.

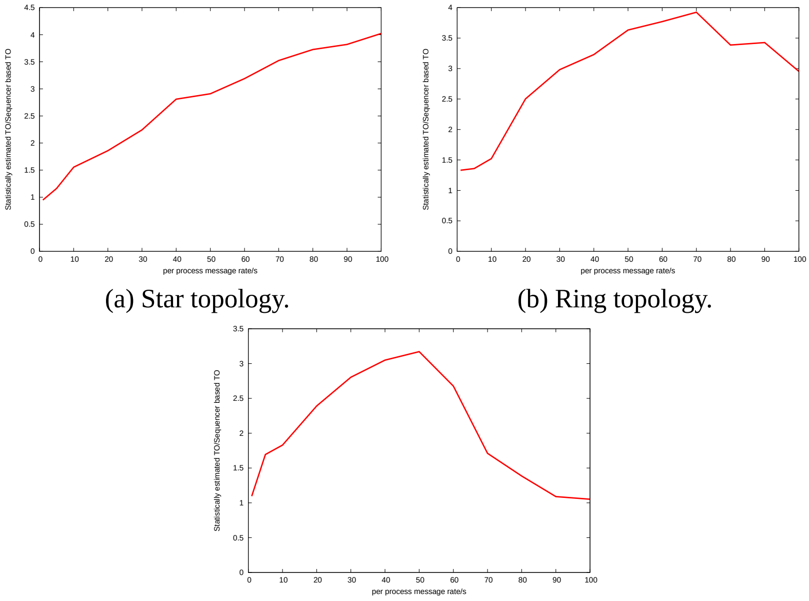
Figure 4.16: Spontaneous total order.

values of spontaneous total order for each topology are depicted in Figure 4.16.

The star topology resembles network topology 1 of the simulation model where the latency between every two processes is 30 ms. As predicted in the simulation (Figure 4.11a), the optimistic protocol presents a much higher spontaneous total order, being higher than 90% for message rates up to 100 messages per second per process, as depicted in Figure 4.16a. The spontaneous total order of the optimistic protocol observed in the Ring topology, and depicted in Figure 4.16b is lower than in the Star topology, but higher than the total order protocol. Even in the Bus topology, depicted in the Figure 4.16c the optimistic protocol present better spontaneous total order than the total order protocol. In this case the values became similar for message rates higher than 80 messages per process.

The gain in spontaneous total order depends on the topology being considered and on the message rate. Figure 4.17 presents, for each of the topologies, the ratio of spontaneous total order between the statistically estimated total order protocol and the sequencer based total order protocol. It can be seen that the statistically estimated total order protocol presents better results than the sequencer based total order algorithm, and in several situations it presents spontaneous total order values 3 to 4 times higher than the total order protocol.

Figures 4.18 and 4.19 compare the optimistic window and protocol latency of both

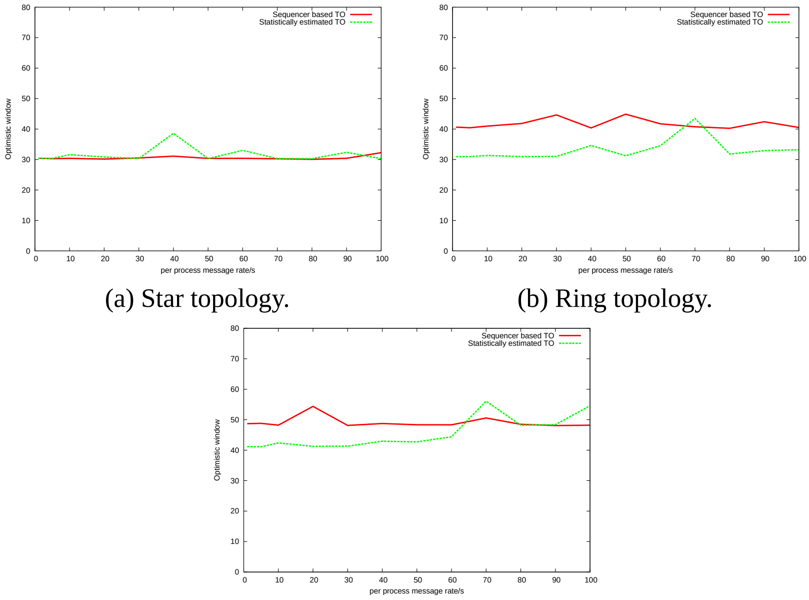


(a) Star topology.

(b) Ring topology.

(c) Bus topology.

Figure 4.17: Comparison of spontaneous total order.

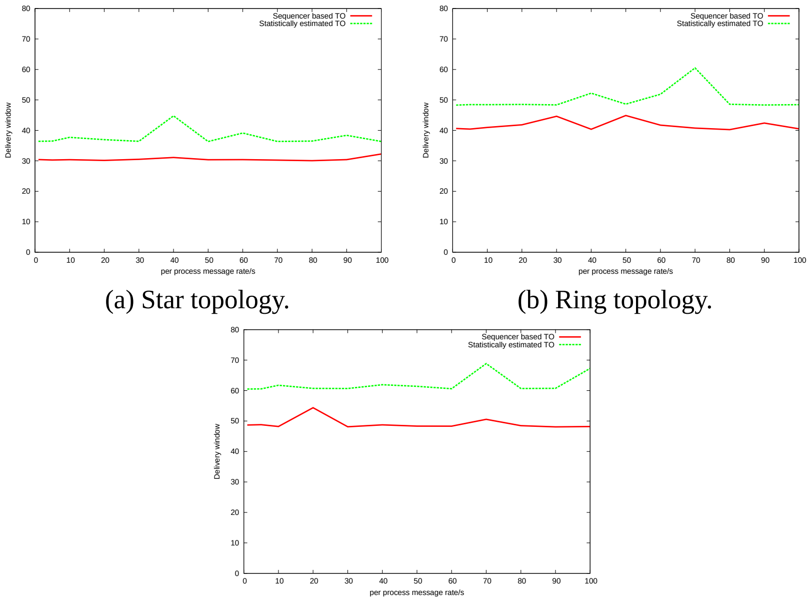


(a) Star topology.

(b) Ring topology.

(c) Bus topology.

Figure 4.18: Optimistic window.



(a) Star topology.

(b) Ring topology.

(c) Bus topology.

Figure 4.19: Delivery window.

protocols in each topology. From the figures it can be observed a small decrease in the optimistic window and an increase in the protocol latency, as expected from the results of the simulation model.

As in the simulation the results demonstrate that the higher degree of spontaneous total order of the optimistic protocol is not obtained sacrificing the optimistic window nor the protocol latency. With respect to the optimistic window, as expected it is smaller as it is being used to improve the spontaneous total order. The protocol latency suffers from the fact that the sequencer is delaying the ordering of its own messages. Being so, there is a small increase in the protocol latency.

4.5 Summary and Open Issues

This chapter proposed a statistically estimated optimistic total order multicast protocol. The protocol is an optimization of the optimistic total order multicast protocol [KPAS99], by improving the spontaneous total order in WANs. The proposed solution only marginally increases latency, due to the delays imposed to the sequencer messages.

The effectiveness of the solution depends on how applications deal with optimistic deliveries and on their reaction to optimistic messages incorrectly delivered. A simple evaluation of its effectiveness can be done as follows.

As an example, consider an application which cannot interrupt the processing of a message after it has started. This means that even if meanwhile there is a final delivery which shows that the optimistic delivery was wrong, it has to wait for it to finish to start the processing of the correct message. Consider also that, the penalty for undoing the effect of the computation when required is negligible. Let g be the time required to process a message. If the latency of final delivery is l and no optimistic processing is done then latency of the whole computation is $l + g$.

Let w be the size of the optimistic window. If $g \leq w$ and there is no penalty for processing messages in incorrect order, then it is obvious that optimistic delivery is useful, as its latency is never worse than the original computation after delivery of the final message. However, if $g > w$ then the effective latency can be either:

- $l + g - w$, when the optimistic delivery is correct;
- $l + g - w + g$, when the optimistic delivery turns out to be wrong.

If r is the ratio of correct deliveries, the average latency is $r(l+g-w)+(1-r)(l+g-w+g)$. Therefore, optimistic delivery decreases latency if $g < w/(1-r)$.

A concern when using an optimistic algorithm is that the performance of the final delivery is not affected, i.e., the optimistic delivery does not increase the algorithm latency. This is not an issue when delaying only optimistic deliveries. In this case, as soon as a sequence number for a message locally available is known it is immediately delivered, even if its optimistic delivery has been erroneously delayed by an excessive amount of time. Nevertheless, it is up to the application to, as soon as possible, interrupt processing of an optimistic delivery if the final delivery happens to be of a different message. A second concern when implementing this technique is the granularity of operating system timers used to delay messages.

The application of the proposed technique to algorithms other than the simple fixed sequencer algorithm presented, is a point that may be looked in the future. This requires that messages are disseminated to receivers before suffering the latency of ordering, excluding algorithms which delay dissemination until messages are ordered [HLvR99, AMMS$^+$95b, ACM95]. It is also required that the decided order is directly derived from the spontaneous ordering at some process, which is not true for causal history algorithms [Lam78, PBS89, EMS95]. These requirements are satisfied by consensus based algorithms [CT96] as long as the coordinator for each instance of consensus is likely to be the same.

Chapter 5

Partial Database Replication

Group based database replication has concentrate a lot of research efforts over the last years [AAAS97, SAA98, PGS98, KPAS99, HAA99, PMJPKA00, KA00a, KA00b, JPPMKA02, AT02, ADMA+02, CMZ04, WK05]. This interest has been motivated by the fact that group communication primitives may be used to ensure transactions' properties [SR96]. Additionally group based replication protocols present promising results regarding the scalability and performance problems of traditional database replication protocols when ensuring strong consistency. The use of group communication primitives allows for efficient data dissemination while reducing the synchronization overheads which are the major problems of traditional database replication protocols [GHOS96].

Common to all group based database replication protocols is the fact they adopt the ROWAA replication protocol [GSC+83], which is the appropriated choice for most of the applications [JPPMAK03].

The ROWAA protocol allows optimal scalability for read operations, as they can be executed locally without interaction with the other database sites. The same scalability for write operations is difficult to achieve, as write operations must be executed by all database sites. This implies that each new replica added to the system do not share the load with the others, as happens with read operations, but instead increases it. Write operations increase the load either at database sites local storage either at the required network bandwidth. The network bandwidth may not be a concern in local area networks, but it is for sure in wide area networks where it is much more limited and expensive.

Applications presenting data access locality, and those like the TPC-C [TPC01] in which data can be easily fragmented, are applications that may reduce the storage and network bandwidth requirements by having each database site replicating only part of the database. This allows to reduce the impact of replication on the storage and network bandwidth requirements. A newer replica added to the system only increases the load on the database sites replicating the same parts of the database and not all database sites, as in the full replication scenario.

The adoption of partial replication poses several challenges to protocols such as the Database State Machine (DBSM) protocol [PGS03], used for full replication.

In a full replication scenario, read operations can be executed by the database site interacting with the client, while write operations require some coordination among database sites. In an optimistic execution, write operations are executed locally by the database site executing the transaction. Coordination between database sites happens only once per transaction before commit.

In a partial replication scenario, local execution of read and write operations is impossible if the database site does not replicates the accessed data items. This requires the characterization of the execution model for partial replication, which depending on the transactions a database site is allowed to execute, may require interaction with other database sites during execution.

In the optimistic execution model, transactions are executed by a database site without interaction with the others until commit time. At that point, transactions must be globally validated, i.e., all database sites must agree on the transaction outcome. This is ensured by a termination protocol responsible to ensure transactions' dissemination, atomicity and consistency properties. In the DBSM, the termination protocol uses an atomic broadcast to disseminate and establish transactions' certification order and a deterministic certification test to ensure transactions atomicity and consistency.

In a partial replication scenario, two approaches have been devised as possible to the termination process. One delivers transaction information to each database site according to the data items replicated by the database site, but requires and additional agreement step. The other is similar to the DBSM approach, but requires more resources from the database sites and possibly more network bandwidth. The adoption of one solution in detriment of the other involves some trade-offs, which are evaluated in the conducted experiences.

This Chapter starts by revising database replication with optimistic execution, and associated database model in Section 5.1. Afterwards, in Section 5.2, it describes the changes and refinements on the components of the database model in order to support partial replication. It proposes two alternatives for the termination protocol and the trade-offs involved in the selection of each of the alternatives. Section 5.3 describes the protocols implementations and finally, Section 5.4 describes the evaluation environment and evaluates the protocols using the TPC-C [TPC01].

5.1 Database Replication with Optimistic Execution

In contrast with replication based on distributed locking and atomic commit protocols, group communication based protocols minimize interaction between replicas and the resulting synchronization overhead by relying on total order multicast to ensure consistency. Generically, the approach builds on the classical replicated state machine [Sch93]: The exact same sequence of update operations is applied to the same initial state, thus producing a consistent replicated output and final state. The problem is then to ensure deterministic processing without overly restricting concurrent execution, which would dramatically reduce throughput, and avoid re-execution in all replicas.

The group based replication protocols of [KPAS99, PMJPKA00, PGS03, KA00a], rely on a totally ordered multicast for consistency. They differ mainly on whether transactions are executed conservatively [KPAS99, PMJPKA00] or optimistically [PGS03, KA00a]. In the former, concurrency is restricted by a priori coordination among the replicas. It is assured that when a transaction executes there is no concurrent conflicting transaction being executed remotely and therefore its success depends entirely on the local database engine. In the latter, execution is optimistic and there is no restriction on where concurrent conflicting transactions are executed. Each replica independently executes its locally submitted transactions and only then, just before committing, sites coordinate and check for conflicts between concurrent transactions.

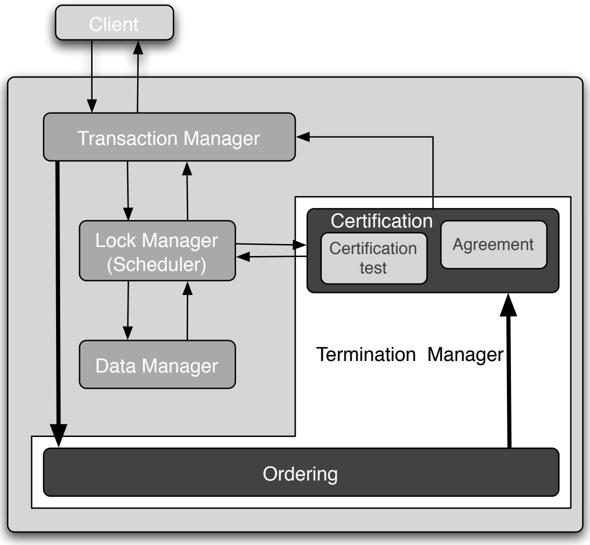


Figure 5.1: Database model.

Database replication with optimistic execution is based on the database system model of [BHG87], augmented with a Termination Manager [PGS03], responsible for the synchronization at commit time. The centralized database system is composed by a Transaction Manager, a Lock Manager (or Scheduler) and a Data Manager, as depicted in Figure 5.1. The Termination Manager is composed by an ordering and a certification module. The first is responsible for the dissemination and ordering of the transactions. The latter is responsible for execution's validity and assuring an agreed outcome.

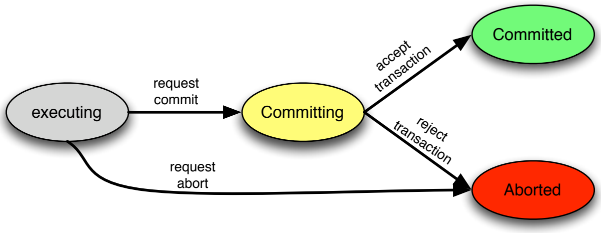


Figure 5.2: Transaction states.

During its lifetime a transaction evolves through some well-defined states (Figure 5.2). It starts by *executing* all operations locally. When the client requests its commitment, the transaction proceeds to the *committing* state. Here, it is ensured that all database sites agree on the transaction's outcome. Lastly the transaction evolves to one of its final states *committed* or *aborted*.

5.1.1 Transaction Manager

The Transaction Manager is responsible for interacting with the Lock Manager in order execute transaction's operations. This is done by forwarding the requested transaction's operations to the Lock Manager that coordinates the local execution according to the database consistency criterion.

Until the reception of the transaction's commit request, transactions are executed locally without interaction with other database sites. When the commit request is received, the transaction information is collected by the Transaction Manager and the coordination task is delegated to the Termination Manager which is responsible for determining the transaction's outcome: *commit* or *abort*. The transaction information collected by the Transaction Manager is of two kinds. Information about the transaction itself, i.e. the data items accessed by the transaction for reading and writing – the Read Set (RS) and Write Set (WS), and the new data produced by the transaction execution – the Write Values (WV). The Termination Manager need also to be provided with information that allow it to establish which transactions executed concurrently.

5.1.2 Lock Manager (Scheduler)

The Lock Manager is the responsible for controlling the local execution of concurrent transactions, i.e. to establish the order in which operations from different transactions are executed.

The goal of the Lock Manager is to, given a set of transactions, produce an execution equivalent to some sequential execution of the same set of transactions. To obtain such result, the Lock Manager allows operations from different transactions to execute concurrently, except when it results in an execution impossible to occur if the transactions were effectively executed sequentially.

The definition of which transaction operations can and which can not be executed concurrently depends on the adopted consistency criterion. The most commonly used consistency criteria are the Serializable consistency criterion (SR) and the Snapshot Isolation consistency criterion (SI) described next.

Serializable Consistency Criterion Serializable consistency ensures that the concurrent execution of a set of transactions is equivalent to some serial execution of the same set of transactions [BHG87]. I.e., a history s is conflict serializable if there exists a serial history s' such that s is **conflict equivalent** to s' [GG02].

Snapshot Isolation Consistency Criterion Snapshot isolation consistency ensures that read operations are never blocked by conflicting transactions, as read operations should always be performed on a snapshot of the (committed) data prior to the transaction's start. Also, update operations should be made on such a snapshot, and will be visible only for transactions starting after the commit of the executing transaction [BBG+95]. A formal definition of snapshot isolation appears in [SWWW00]:

- A multiversion history of transactions $T = t_1, ..., t_n$ satisfies the criterion of snapshot isolation (SI) if the following two conditions hold:

SI-V SI version function: The version function maps each read action $r_i(x)$ of t_i to the most recent committed write action $w_j(x)$ as of the time of the begin of t_i, or more formally: $r_i(x)$ is mapped to $w_j(x)$ such that $w_j(x) < c_j < b_i < r_i(x)$ and there are no other actions $w_h(x)$ and $c_h (h \neq j)$ with $w_h(x) < b_i$ and $c_j < c_h < b_i$.

SI-W disjoint write-sets: The write-sets of two concurrent transactions are disjoint, or more formally: if, for two transactions t_i and t_j, either $b_i < b_j < c_i$ or $b_j < b_i < c_j$, then t_i and t_j must not write a common object x.

5.1.3 Data Manager

The Data Manager is the component responsible for reading and writing data from and to stable storage. It ensures the durability of the data produced by transactions processing. Upon commit, the transactions changes to the database are stored in stable storage and survive database failures.

5.1.4 Termination Manager

The transaction's termination, handled by the Termination Manager, is started by the Transaction Manager when receiving the transaction's commit request. The transaction information and database state received from the Transaction Manager are propagated to all database sites. Each database site will then *certify* and, if possible, commit the transaction. The Termination Manager has thus three goals: *i)* to propagate the transaction to all database sites, *ii)* to certify it, and *iii)* to commit it.

5.1.4.1 Transaction's Dissemination and Ordering

The first step of the Termination protocol is to propagate transaction information to all database sites, so it can be certified and committed at all replicas. After receiving the transactions, each database site has to establish their certification order.

The atomic broadcast is usually the protocol of choice for establishing a global order among a set of transactions. Atomic broadcast blindly orders transactions independently of being or not conflicting transactions.

The generic broadcast [PS99] and the reordering certification test [Ped99] are two technics that can be used in order to improve the number of committed transactions. The former orders transactions based on a conflict relation. The conflict relation is defined by the application, and is used to establish an ordering among conflicting transactions. The latter, uses an atomic broadcast protocol to establish the certification order, and a fixed size reorder list of certified transactions not yet committed. This list allows to set different certification and commit orders for each transaction by changing their order in the list. I.e., if the result of a transaction certification is abort but would be commit if the transaction was certified before some of the transactions in the reorder list, then it will be inserted in the reorder list before the transactions causing it to abort, thus improving the number of committed transactions.

5.1.4.2 Transaction Certification

Transaction certification is based on the principle of *certifiers* [BHG87]. A certifier is an optimistic scheduler that never delays operations, but that from time to time verifies whether the operations it had allowed to execute do not violate the database consistency criteria. This verification has to be done before committing any transaction, possible of leading the database to an inconsistent state.

In order to be committed, a transaction, t, can not conflict with already committed transactions. The aim of the certification procedure is thus to ensure that t do not conflicts with already committed transactions, t'. This is done by: *i)* establishing which committed transactions t' are concurrent with t, and which precede it. A transaction t' precedes a transaction t if t' commits before t start; *ii)* ensuring there are no operation conflicts between t and concurrent transactions t' already committed.

The definition of the certification procedures for the consistency criteria of Section 5.1.2 are presented below, as the definition of operation conflicts depends on the adopted consistency criteria

Serializable Certification In a database using serializable consistency criterion, two operations conflict when they are issued by different transactions, access the same data item and at least one of them is a write operation. Thus, operations conflicts can be read-write, when one transaction reads a data item written by the other, or write-write, when both transactions write the same data item.

In such a database, a transaction only reaches the commit request if it do not conflicts with transactions committed during its optimistic execution. In the event of a conflict the transaction should have been aborted.

This fact can be used to reduce the set of concurrent transactions that must be checked for conflicts by the certification procedure. Transactions, t', committed during the optimistic execution of t, can be considered as preceding it. If they were conflicting then the transaction must have been aborted during its optimistic execution. This is the precedence relation between transactions t and t' appearing in [PGS03]: (*i*) if t and t' execute at the same database site, t' precedes t if t' enters the committing state before t; or (*ii*) if t and t' execute at different database sites, for example s_i and s_j, respectively, t' precedes t if t' commits at s_i before t enters the committing state at s_i.

In order to certify a transaction t the certification procedure has to ensure that every committed transaction t' either precedes t, or there are no read-write nor write-write conflicts between the read and write sets of t and the write set of t'.

Snapshot Isolation Certification In a database using snapshot isolation as the consistency criterion, transaction's t read operations of a data item always return the same value, independently of the data item being updated by concurrent transactions during t's execution. In such a database, read operations never conflict with write operations.

From the snapshot isolation consistency criteria definition, two write operations are conflicting if, issued by concurrent transactions and update the same data item. Being so, in order to certify a transaction t the certification procedure has to ensure that every committed transaction t' either precedes t, or their write sets are disjoint.

5.1.4.3 Distributed Agreement

Having all database sites deterministically certifying transactions may not be sufficient to ensure the replicas' consistency. After certification there may be some local constraints preventing a database site to commit a transaction.

The problem of ensuring that a group of database sites reach a common decision on whether to commit or abort a transaction might require the use of an atomic commitment protocol. When the protocol has to ensure that every participant reaches a decision despite the failure of other participants, this atomic commitment protocol is called Non-Blocking Atomic Commitment protocol (NB-AC) [BHG87].

In an asynchronous distributed system, where perfect failure detection is not available, a weaker version of the protocol is required. Roughly, the Weak Non-Blocking Atomic Commit protocol [Gue95], leads to a commit decision when all database sites propose to do so and no one is suspected to have failed.

To avoid aborting transactions due to constraints affecting a single database site the atomic commitment may be replaced by a dictatorial atomic commit [AGP02], a protocol in which database sites cannot propose to abort a transaction, but must decide accordingly to the coordinator will. Disallowing a database site to unilaterally abort transactions, i.e. ensuring that after successfully certifying a transaction a database site cannot unilaterally abort it, the atomic commitment may be avoided. In this case all database sites will propose the same transaction outcome, and the result of the atomic commitment can only be the proposed value, rendering it redundant.

In both cases, a database site failing to comply with the decision implies that the database site quits and leaves the group.

5.1.4.4 High Priority Transactions

After successful completion of the termination process, the transaction outcome is commit, so all its updates must be applied to all database sites. During the update process, the Lock Manager must ensure that: *i)* the transaction will not be aborted due to conflicts with other executing transactions; *ii)* transaction updates are applied to the database in certification order.

The first requirement dictates that certified transactions have priority over locally running transactions. This means, that whenever a conflict is detected by the local Lock Manager it can only decide to abort the locally running transaction.

The second requirement dictates how conflicts between certified transactions are handled. In this case, the conflict results from both transactions trying to update the same data item. The solution is thus to ensure that updates are applied to the database in the certification order, or else that only the updates resulting from the latest certified transaction are reflected in the database.

5.2 Partial Replication

This section proposes a group based partial database replication protocol with optimistic execution. It follows the structure of the previous one describing the changes and refinements necessary in some of the modules to support partial replication.

Full and partial replication differ on the definition of which data items are replicated in each database site. The definition of the database sites replicating each data item is orthogonal to the replication protocol. It must be aware of which data items are replicated at each database site, in order to establish which database sites are involved in each transaction. For instance, in a relational database fragmentation might be table based, or based on the horizontal or vertical fragmentation of tables.

The partial replication execution model may require some changes to the Transaction Manager to support it. Changes are required if the execution model does not restrict database sites to execute transactions to access only data items replicated locally. The changes are required in order to allow the Transaction Manager to coordinate the transactions' optimistic execution with other database sites. These changes to the Transaction Manager are discussed in Section 5.2.1.

The Termination Manager, faces several challenges. It has to solve the problem of delivering transaction's information only to database sites replicating it, which means delivering parts of the transaction information to different database sites. It has to establish if transactions should be ordered before certification and the necessity of an atomic commitment after certification.

To solve the challenges faced by the Termination Manager, we envision two approaches. Both require write values to be disseminated apart from the ordering protocol, in order to reduce the required network bandwidth. Regarding read and write sets dissemination, one of the approaches uses the one of the previous Section. The other requires an atomic commitment after certification, uses an atomic broadcast to define the transactions certification order and a reliable broadcast to concurrently distribute parts of the read and write sets to the database sites replicating them.

There are some trade-offs involved in the selection of one of the protocols in detriment of the other, which are presented in Section 5.2.2.

5.2.1 Transaction Manager

The Transaction Manager, as in the full replication scenario is the responsible for coordinating the transactions' execution. The main difference compared to the full replication is that some transactions can not be executed locally. That is, the Transaction Manager has to deal with the fact that the database site only replicates part of the database.

To deal with this problem the Transaction Manager may follow two distinct approaches. The first is to ignore the part of the database not replicated locally. Using this approach transactions running locally only access data items for which the database site holds a replica. Using this approach there is no difference between the Transaction Manager of a fully or partially replicated database.

Alternatively, the Transaction Manager is aware that there are data items replicated elsewhere. In this case, whenever a transaction tries to access these data items the local Transaction Manager must ensure the transaction accesses them. This problem occurs only during the optimistic execution and has been the subject of the work in [Jún04]. The proposed solutions are briefly described below.

The referred work adopted a distributed query processing mechanism which uses a two step-optimization to solve the problem of distributed execution. Roughly, the two step-optimization facilitates the integration of the distributed query processing into a centralized database management system, allowing the query to be locally optimized without further modifications to the local optimization engine. After the local optimization, which corresponds to the first step, the second step must decide where the pre-processed query must have its operations executed. In this case, for each operation, a simple approach which chooses the first database site that is able to handle the request has been adopted.

Combined with the two-step optimization, a distributed execution that mimics a nested transaction has been developed: (i) the initiator (i.e., the database site used by the client to send transaction's request) can distribute one sub-transaction per database site; (ii) only the initiator can distribute sub-transactions, which avoids the possibilities of deadlocks inside the same transaction; (iii) sub-transactions execute optimistically at remote database sites and the concurrency control mechanism of the initiator database site controls its own transactions; (iv) upon the initiator abort, all the sub-transactions are also aborted; (v) upon a sub-transaction abort, all the other sub-transactions are also aborted.

In order to minimize the impact of the distributed execution on the overall performance, which may increase resource usage and mainly bandwidth consumption, a distributed cache mechanism was built. It is based on semantic entries, which means that instead of using tuples or pages to identify the entries in cache, the predicates of the queries are used as identifiers and the cache is populated using the results of the queries. This avoided the management overhead of the tuples, which usually involves retrieval, update and replacement per tuple.

The described protocol is easily integrated in the described architecture. Being fully integrated in the Transaction Manager allows it to be used transparently, as it does not imply changes in the validation process.

5.2.2 Termination Manager

In partial replication there are data items that are replicated only at some of the replicas. Being so, it is natural that database sites are only interested in operations referring to data items replicated locally. This poses a challenge to the dissemination which must decide which parts of the read and write sets should be sent to each database site.

There are two options regarding read and write sets dissemination: selectively send to each database site parts of the read and write set containing the data items it replicates; or send the read and write sets to every replica as it happens in full replication. We analyze both approaches next.

5.2.2.1 Partial Certification

In this scenario, each database site delivering a transaction only receives the part of the read and write sets and the write values relating to the data items replicated locally. The received read and write sets only allows each database site to certify part of the transaction, i.e. the part regarding the data items it replicates. Having such an incomplete knowledge each database site can not decide the transaction outcome without consulting the other database sites.

The need for a final agreement after the transaction's certification leads to question the utility of using a protocol ensuring a global order in the beginning of the termination process. It turns out, however, that ordering transactions before certifying them improves the ratio of committed transactions [PGS98].

The total order multicast to multiple groups protocol [GS97a], ensures that: *i)* only database sites replicating the data items accessed by a transaction deliver the transaction; and *ii)* transactions are delivered in the same order by all database sites replicating data items accessed by it.

This protocol fulfills all the requirements of transaction's dissemination. The major drawback is that implementing such a protocol is costly and the genuine multicast version is usually not implemented in group communication toolkits. In this case, the option is to use an atomic multicast to all database sites. This is an undesired situation as with partial replication we intend to reduce the required network bandwidth.

To reduce the bandwidth required by the atomic multicast protocol we adopt a protocol that is the combination of two distinct protocols. The first orders transactions among all database sites, and the second uses a reliable multicast to send to each database site only the parts of the transaction information referring data items replicated locally. These two protocols run concurrently and their impact on required network bandwidth should be marginal.

Using a global order to certify transactions, results in all replicas of a data item x, $Sites(x)$, detecting the same conflicts. Requiring database sites to propose the result of the certification for the atomic commitment, leads to, all database sites replicating conflicting data items to propose *no*, and all the other proposing *yes*.

From this, after receiving *yes* votes from database sites representing all data items accessed by a transaction t, $Items(t)$, a participant in the atomic commitment protocol knows that decision will be commit. At that point a participant knows that the transaction passes the certification test in all data items, and that the remaining votes must be *yes* because the participants are only allowed to propose the result of the certification. Since, after collecting votes from a representative of each data item all the remaining votes are redundant, it should be possible to tolerate the failure of some database sites, without affecting the transaction's outcome.

The weak non-blocking atomic commit decides to abort a transaction if a during the protocol execution a database site is suspected of having failed. To circumvent this, and taking advantage of the fact that we should tolerate the failure of some database sites, we propose the use of a new definition for the atomic commitment problem, the *Resilient Atomic Commit (RAC)*. Resilient atomic commit is an atomic commit protocol able to decide commit as long as there is a replica of every data item that is not suspected of having crashed. In this protocol, participants start by voting *yes* or *no* for the outcome of transaction t. The result should fulfill the following conditions:

Termination. Every correct participant eventually decides.

Agreement. No two participants decide differently.

Validity: If a participant decides *commit* for t, then for each $x \in Items(t)$, there is at least a participant in $Sites(x)$ that voted *yes* for t.

Non-triviality: If for each $x \in Items(t)$ there is at least a participant $s \in Sites(x)$ that votes *yes* for t and is not suspected, then every correct participant eventually decides *commit* for t.

Figure 5.3 depicts the execution of transaction t, which is committed using resilient atomic commit but aborted if using atomic commit. In step 1, transaction t executes at database site s_1, and client c sends a commit request to the database site s_1. In step 2, t is broadcast and at the end of this step, it is delivered, certified and s_2 crashes. Sites s_1 and s_3 start the Resilient Atomic Commit protocol voting *yes* and using s_1 as coordinator, which decides *commit* at the end of step 3 (using Atomic Commit, the transaction will be aborted since s_2 is eventually suspected to have failed). In step 4, s_1 sends its decision to all database sites. In step 5, database sites s_1 and s_3 receive the decision of the Resilient Atomic Commit and s_1 sends the transaction result to c.

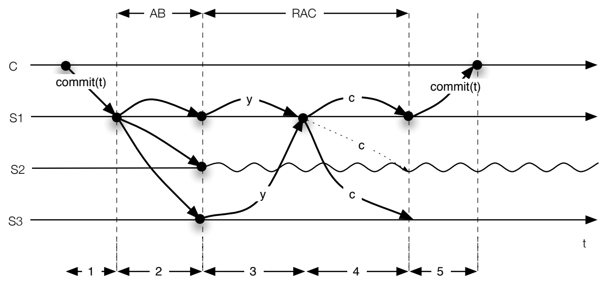


Figure 5.3: Validation with Resilient Atomic Commit.

Resilient Atomic Commit presents better latencies than the Atomic Commit protocol. The Atomic Commit protocol has to wait until a vote is received from each participant or until it is suspected of having failed to decide. On the other hand, the Resilient Atomic Commit can decide when the collected votes are representative of all data items accessed by the transaction. This particularity of the Resilient Atomic Commit represents significant improvements in wide area networks, or when a database site becomes much slower than the others. The latency of the Resilient Atomic Commit will be equal to the Atomic Commit protocol only when the farthest or slower database site holds the only replica of certain data items.

5.2.2.2 Independent Certification

In a scenario in which the read and write sets are disseminated with the atomic multicast, every database site receive the transaction's read and write sets, independently of the data items it replicates.

As in the previous scenario, the write values are only sent to the replicas of each data item using a reliable multicast.

Having the read and write sets sent to all replicas is a scenario similar to full replication, i.e., every database site as the necessary information to certify the transaction, and the final atomic commitment can be avoided.

This scenario establishes some trade-offs with the previous one, namely on the additional bandwidth and memory required for transmitting the full read and write sets and storing the full write sets, as well as the latency of the termination process. These trade-off are discussed in the following section.

5.2.2.3 Trade-offs

On the choice of the termination approach to use, both network and memory usage need to be considered.

Network Bandwidth In order to evaluate the tradeoff involving the bandwidth required by both approaches, the transactions' read set, write set, and write values have been divided in two subsets: *i)* a subset of data items replicated locally called RS_L, WS_L and WV_L, and *ii)* a subset containing data items not replicated locally called RS_R, WS_R. AC represents the amount of data exchanged among the replicas during the execution of the atomic commitment protocol.

In a system with n replicas the required bandwidth for each case is given by:

$$\text{Full RS \& WS} \equiv \sum_{i=1}^{n-1}(RS_{L_i} + WS_{L_i} + WV_{L_i}) + \sum_{i=1}^{n-1}(RS_{R_i} + WS_{R_i}) \tag{5.1}$$

$$\text{Partial RS \& WS} \equiv \sum_{i=1}^{n-1}(RS_{L_i} + WS_{L_i} + WV_{L_i}) + AC \tag{5.2}$$

From 5.1 and 5.2 in terms of bandwidth requirements, selective read and write sets are preferable as long as the required bandwidth for the atomic commitment protocol does not exceed the required for transmitting the missing values of the read and write sets, i.e. as long as

$$AC < \sum_{i=1}^{n-1}(RS_{R_i} + WS_{R_i}) \tag{5.3}$$

When the adopted consistency criteria is snapshot isolation, since the certification test does not depend on the read set, the previous expression can be rewritten as:

$$AC < \sum_{i=1}^{n-1}(WS_{R_i}) \tag{5.4}$$

Latency of the Termination Process The different protocols used during the termination process as well as the size of the data being transferred should result in different latencies among the different approaches.

The contributors for the latency of the termination process are the atomic broadcast latency and the certification latency. Additionally, when sending only parts of the read set to each database site, the termination process also incurs in the latency of the atomic commitment.

Regarding the atomic broadcast protocol, as this protocol propagates the messages concurrently with the ordering mechanism, it is expected that it will mask the differences in latency that should happen due to message size.

Regarding the certification test, the main difference among the two situations is on the size of the read and write set. It is expected that the certification test only marginally contributes to the overall latency.

The factor that should negatively impact the latency of the termination is the atomic commitment protocol, and in this case the protocol sending full read and write sets should be in advantage.

Memory Usage As in the case of the bandwidth, sending the read set and write set to all database sites, requires additional memory at each database site. This additional memory is required in order to hold the parts of the write sets referring to data items not replicated at the database site.

The evaluation presented in Section 5.4.4 will provide concrete figures for the involved trade-offs.

5.3 Implementation

This Section describes the implementation of the of the partial replication termination protocols. It starts with a description of the implementation architecture and of the interfaces used by the termination protocols to interact with the database and group communication.

Afterwards, Section 5.3.3 describes the class implementing transactions' certification, which is used by the implementation of the termination protocols.

Section 5.3.4 describes the termination protocol with independent certification. The refinements required to support optimistic delivery are also described. The same protocol implementation can be used in the full replication scenario.

Finally, Section 5.3.5 describes the implementation termination protocol with partial certification, as well as the implementation of the resilient atomic commitment protocol.

5.3.1 Architecture

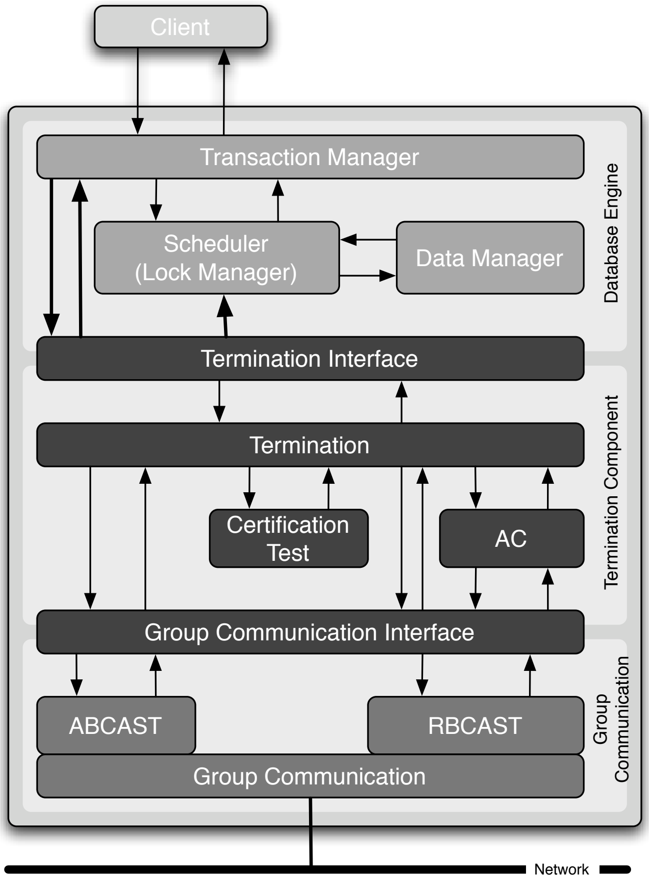


Figure 5.4: Implementation architecture.

The system architecture used for the implementation of the replication protocols is depicted in Figure 5.4. The architecture presents three major blocks, the database engine, the termination component and group communication. Communication between the database engine and the termination component is mediated by the termination interface and between the termination component and the group communication by the group communication interface.

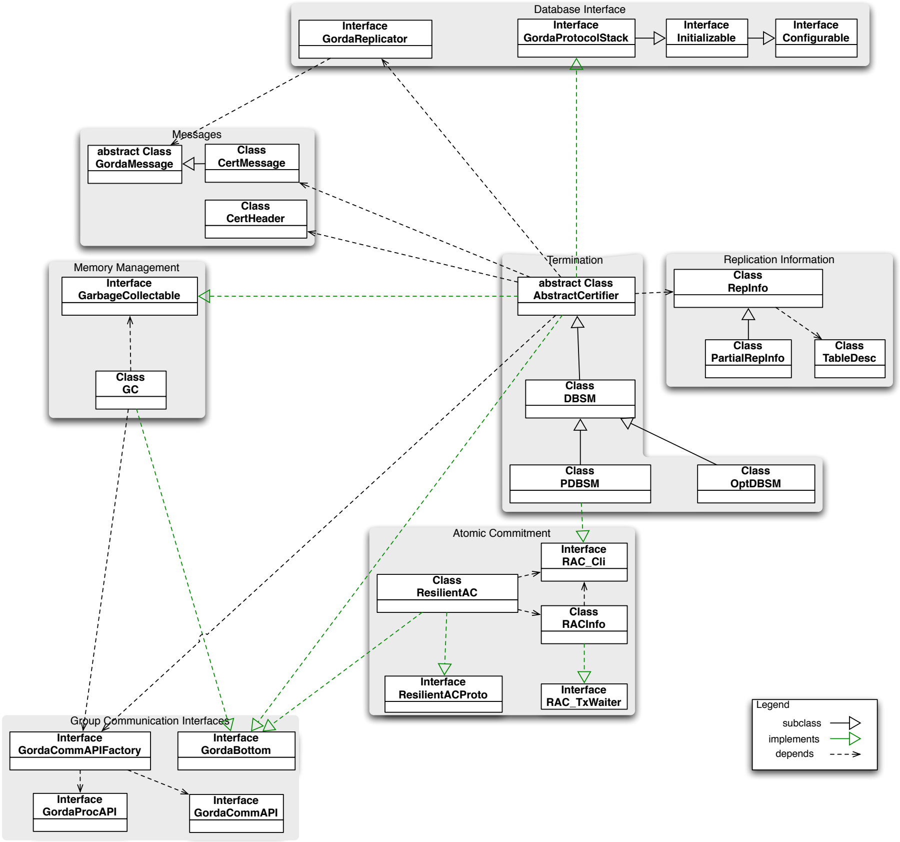


Figure 5.5: Classes used in the implementation.

The classes and interfaces of the JAVA implementation of the presented architecture are depicted in Figure 5.5. It presents four main components, the database interface, the group communication interface, the termination component and the atomic commitment component. It presents also three auxiliary components, the replication information, messages and memory management components.

The database and group communication interfaces define the interfaces for interaction with the termination component.

The termination component defines an abstract class, the `AbstractCertifier` which implements the certification test and is the base for the implementation of the termination protocols.

The atomic commitment module defines interfaces and implements the resilient atomic commitment protocol.

The replication information module is used to store information about replicated data, for instance the location of the replicas of data items.

The memory management module is responsible to exchange and collect information about committed transactions, so memory occupied by transactions preceding any running transaction can be freed.

The messages module defines the data formats of the information exchanged between the database and the termination protocol, as well of the messages exchanged during the termination protocol.

5.3.2 Termination and Group Communication Interfaces

The interface between the database and the termination module is ensured by the four interfaces depicted in Figure 5.6. The database implements the interface `GordaReplicator` which defines the `deliver` method for the termination process to send transactions originated remotely to be executed, and to commit or abort transactions originated locally. This interface defines also the method `getOldestSnapShotTime` to be used by the termination protocol to query the database about the version of the oldest snapshot still in use. This is used to establish which committed transactions can be purged from memory by the termination process.

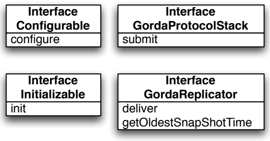


Figure 5.6: Database interface.

The `GordaProtocolStack` interface is implemented by the termination protocol and defines the method `submit` used by the database to trigger the termination process. The termination protocol implements the `Configurable` and `Initializable` interfaces on which `GordaProtocolStack` depends. These are used to handle information about replicated data and to initialize the instance of termination protocol.

The group communication to termination protocol interface is ensured by the interfaces depicted in Figure 5.7. the `GordaBottom` interface, which defines methods for view management `flush` and `install`, `unblockMulticast` for flow control and for message exchange `deliver` and `fastDeliver`, is implemented by the termination process. The group communication implements the `GordaCommAPI` interface which defines the counterpart methods for view management `done`, for flow control `unblockDeliver` and for message exchange `multicast`.

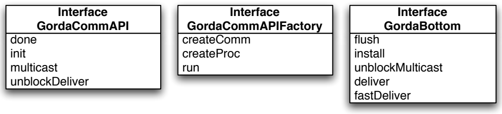


Figure 5.7: Group communication interface.

The `flush` method of `GordaBottom` is called before changing a view in order to deliver the messages that should be delivered in the current view. When this task is concluded, then the `done` method in the `GordaCommAPI` should be called. The `install` is used to change the view due to a change in the group membership.

The `unblockMulticast` is used to inform a `GordaBottom` that it can resume the sending of messages, which must have stopped after a multicast returning `false`.

The `deliver` and `fastDeliver` are used to deliver to the termination protocol a message when the required properties are achieved or to deliver it optimistically. One of such calls returning the value of `false` prevents the group communication to deliver further messages until it receives a `unblockDeliver` .

The message delivery guarantees offered by the group communication may range from reliable to uniform total order. Currently, the ones used are reliable, total order and optimistic total order. The interface offered by the group communication to the termination protocol, the `GordaCommAPIFactory` allows the termination protocol to establish the desired properties when creating an instance of a communication channel by calling the `createComm` method. A communication channel created by the `GordaCommAPIFactory` implements the `GordaCommAPI` .

5.3.3 Abstract Certification

The abstract class `AbstractCertifier` is the basic building block of the termination protocol. It implements the certification tests and interfaces with the database. The serializable and snapshot isolation certification tests are implemented in the procedures `certification_sr` and `certification_si` depicted in Listing 5.1 and 5.3 respectively.

Depending on the transactions profile, the read sets may be quite large. Transmitting these large read sets over high latency and low bandwidth links may be problematic. To alleviate the replication protocol of transmitting such amounts of information, applications may establish thresholds on the number of read items from a relation. When such a threshold is reached, then the table is considered as being fully read, which results in sending only a data item, representing the whole table, over the network.

```
1   boolean certCheckSR(CertMessage curr, CertMessage[] conc){
      long[] rTablesLocked = curr.rtLocks();
      long[] rs = curr.RS();
      for (int i = 0; i < conc.length; i++) {
5         if (rTablesLocked != null && rTablesLocked.length > 0)
            if(intersects(rTablesLocked, conc[i].wTables(), 0,rTablesLocked.length, 0,
                (conc[i].wTables() != null ? conc[i].wTables().length : 0)))
              return false;
          if(intersects(rs, conc[i].WS(), 0, rs.length, 0, (conc[i].WS() != null ? conc[i].WS().length : 0)))
10          return false;
        }
    }

    boolean certification_sr(CertMessage curr, CertMessage[] finalc, CertMessage[] fastc) {
15      if (finalc != null && !certCheckSR(curr,finalc))
          return false;
        if (fastc != null &&!certCheckSR(curr,fastc))
          return false;
        return true;
20  }
```

Listing 5.1: Serializable certification test.

The serializable certification test starts to check for conflicts against concurrent transactions already committed (line 15) and against optimistically delivered transactions passing the certification test (line 17), when using optimistic delivery of the transactions, using the `certCheckSR` function. This function starts to check, for each concurrent transaction, if due to its size the read set is a full table, and looks for conflicts between tables (lines 5-7). Afterwards it looks for conflicts on individual data items (line 9).

For conflict search, the certification test uses the `intersects` procedure which looks for intersections between sets of sorted data items. This procedure is depicted in Listing 5.2 and receives as arguments, two ordered arrays of data item identifiers, and the boundaries for the search of conflicts. It returns a boolean value indicating whether a conflict has been found.

The snapshot isolation certification test is simpler than the serializable certification test as it only has to look for conflicts on written data items. As the previous one, it recurs to the intersects procedure depicted in Listing 5.2.

```
boolean intersects(long[] src1, long[] src2, int frst1, int lst1, int frst2, int lst2) {
  while (frst1 < lst1 && frst2 < lst2)
    if (src1[frst1] < src2[frst2])
      ++frst1;
    else if (src2[frst2] < src1[frst1])
      ++frst2;
    else
      return true;
  return false;
}
```

Listing 5.2: Check for conflicts on data items.

```
boolean certCheckSI(CertMessage curr, CertMessage[] conc){
  long[] ws = curr.WS();
  for (int i = 0; i < conc.length; i++)
    if (conc[i] != null && intersects(ws, conc[i].WS(), 0, ws.length, 0, (conc[i].WS() != null ? conc[i].WS().length : 0)))
      return false;
}

boolean certification_si(CertMessage curr, CertMessage[] finalc, CertMessage[] fastc) {
    if (finalc != null && !certCheckSI(curr,finalc))
        return false;
    if (fastc != null && !certCheckSI(curr,fastc))
        return false;
    return true;
}
```

Listing 5.3: Snapshot isolation certification test.

The auxiliary classes used by the certification test and the termination protocol are the `CertMessage` that holds transaction's information, the `RepInfo` holding information regarding tables identifiers, and replication maps in the case of partial replication and `certMessage` manipulation functions.

5.3.4 Termination Protocol with Independent Certification

The rational behind the implementation of the termination protocol with independent certification, i.e., with read and write sets sent to all replicas is depicted in Figure 5.8. The process relies on four queues, the `LocalTransactions` queue for transactions submitted by the local database, the `FNLDelivered` queue for transactions whose total order has been established, the `Terminated` queue for transactions whose outcome is already known, and the `Committed` queue for transactions that have been committed and are still needed for certification.

The termination process for a transaction is triggered by calling the `submit` procedure. This in turn places the transaction in the queue of `LocalTransactions`

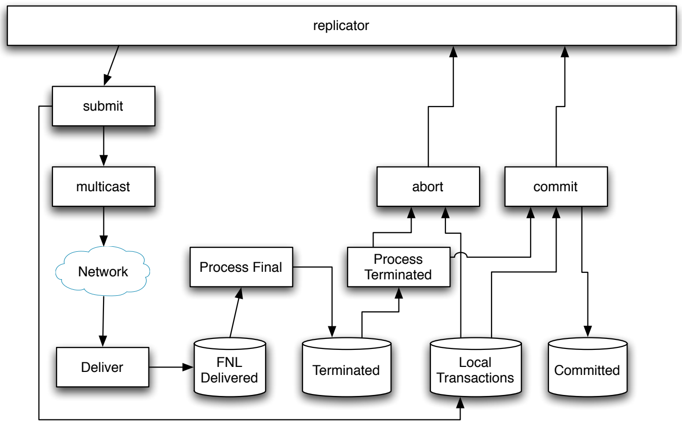


Figure 5.8: Termination protocol with independent certification.

and multicast it to all database sites. When received totally ordered from the network the transaction is added to the end of the `FNLDelivered` queue. A transaction is considered ready to be delivered when both its order, and remaining information, i.e., the read set, the write set, the write values, and precedence information has been received.

The existence of transactions in the `FNLDelivered` queue triggers the execution of the `processFinal` procedure, depicted in Listing 5.4. It starts removing the information at the head of the `FNLDelivered` queue (line 2), afterwards it obtains the transaction information and certifies it (lines 3-5). After certification, the transaction identification and certification outcome are added to the `Terminated` queue (lines 6-12).

```
 1  void processFinal() {
      Object tid = FNLDelivered.remove(0);
      CertHeader ctinf = (CertHeader) tid2tx.get(tid);
      boolean res = certification(ctinf.payload(), build_prevs(
 5            ctinf.payload().getSnapshotTime()));
      CertWrapper<CertHeader> toTerm = new CertWrapper<CertHeader>(ctinf);
      toTerm.setFinal();
      if (res)
        toTerm.setCommit();
10    else
        toTerm.setAbort();
      Terminated.add(toTerm);
    }
```

Listing 5.4: processFinal procedure.

The processTerminated procedure, depicted in Listing 5.5, is triggered by the existence of transactions in the Terminated queue. It starts removing the transaction at the head of the queue (line 2), and depending on the certification result it calls the commit (line 4) or abort (line 6) procedure, depicted in Listing 5.6.

```
1   void processTerminated() {
      CertWrapper<CertHeader> tx = Terminated.remove(0);
      if (tx.isCommit())
        commit(tx.getWrapped());
5     else
        abort(tx.getWrapped());
    }
```

Listing 5.5: processTerminated procedures.

```
1   protected void commit(CertHeader tx) {
      localTrans.remove(tx.tid());
      tx.payload().setAction(ACTION_FINAL_COMMIT);
      replicator.deliver(tx.payload());
5
      tx.payload().setCommitTime(dbVersion);
      Committed.add(tx.payload());
      dbVersion++;
    }
10
    void abort(CertHeader tx) {
      localTrans().remove(tx.tid());
      tx.payload().setAction(ACTION_FINAL_ABORT);
      replicator.deliver(tx.payload());
15
      tid2tx.remove(tx.tid());
    }
```

Listing 5.6: commit and abort procedures.

The commit procedure starts by removing from the LocalTransactions queue the transaction being committed (line 2). The previous operation is only relevant for transactions for which the current database site has triggered the termination process. Afterwards, the commit procedure calls the database to inform it of the transaction outcome (lines 3-4), sets the transaction commit time used for certification, ads it to the Committed queue and increments the local database number (lines 6-8). The abort procedure starts by removing the transaction from the LocalTransactions queue (line 12). Afterwards it calls the database to inform it of the transaction outcome (lines 13-14). Finally it removes the remaining references to the transaction (line 16).

5.3.4.1 Optimistic Total Order

The use of optimistic total order requires some changes on the protocol described earlier, namely for process optimistically delivered transactions. These changes are depicted in Figure 5.9.

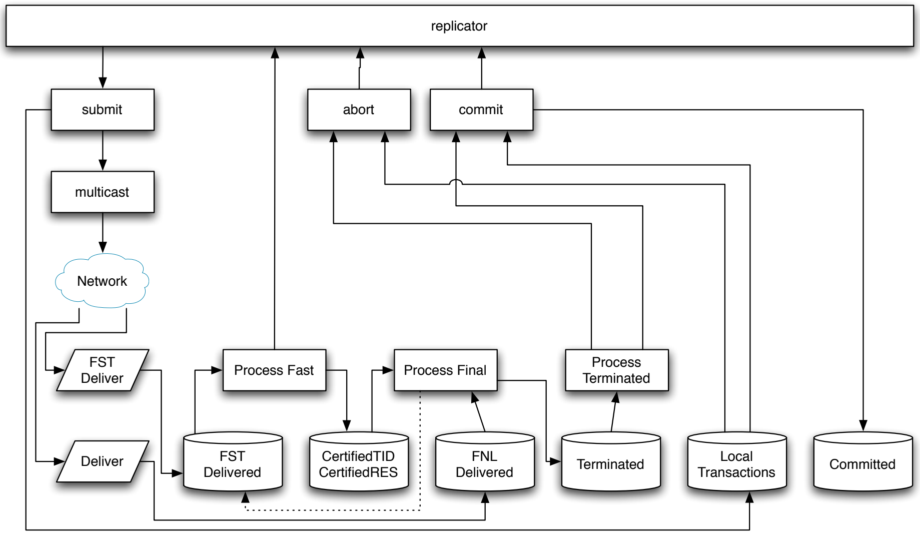


Figure 5.9: Termination protocol with optimistic total order.

In the implementation the changes resulted in the queue `FSTDelivered` for transactions being optimistically delivered and the `Certified` queue for transactions certified optimistically. The processing of optimistically delivered transactions is handled by procedure `processFast` depicted in Listing 5.7. Once again, the transaction is only considered after all its information have been received.

```
void processFast() {
  Long tid = .remove(0);
  CertHeader tx = (CertHeader) tid2tx.get(tid);
  boolean res = super.certification(tx.payload(), build_prevs(tx.payload().getSnapshotTime()), build_fastprevs());
  CertifiedTID.add(tid);
  CertifiedRES.add(new Boolean(res));
  if(res){
    tx.payload().setAction(ACTION_FAST_COMMIT);
    replicator.deliver(tx.payload());
  }
}
```

Listing 5.7: processFast procedure.

The `processFast` procedure starts by removing the transactions at the head of the `FSTDelivered` queue, obtains the transaction information and certifies it (lines 2-5). After certification it inserts the transaction identifier and the certification result into the `Certified` queue (lines 6-7). If the result of the certification is commit, then the database is informed of the possibility of committing the transaction (lines 8-11).

```
1   void processFinal() {
      Object tid = FNLDelivered.get(0);
      if (!CertifiedTID.isEmpty() && tid.equals(CertifiedTID.get(0))){
        FNLDelivered.remove(0);
5       CertifiedTID.remove(0);
        Boolean res=CertifiedRES.remove(0);
        CertWrapper<CertHeader> toTerm = new CertWrapper<CertHeader>((CertHeader) tid2tx.get(tid));
        toTerm.setFinal();
        if (res.booleanValue())
10        toTerm.setCommit();
        else
          toTerm.setAbort();
        Terminated.add(toTerm);
      }else{
15      super.processFinal();
        CertifiedTID.addAll(FSTDelivered);
        FSTDelivered(CertifiedTID);
        FSTDelivered.remove(tid);
        CertifiedTID(new LinkedList<Long>());
20      CertifiedRES(new LinkedList<Boolean>());
      }
    }
```

Listing 5.8: processFinal procedure.

When the authoritative order for the transaction is received the `processFinal` procedure depicted in Listing 5.8 is called. Depending on whether the transaction in the head of the `Certified` queue is also the transaction in the head of `FNLDelivered` queue the procedure executes the code corresponding to a transaction whose optimistic and authoritative orders match (lines 4-13) or the code corresponding to a transaction whose optimistic and authoritative orders do not match (lines 15-20). In the first case the references for the transaction are removed from the `FNLDelivered` and `Certified` queues (lines 4-6) and the transaction information and outcome are added to the `Terminated` queue (lines 7-13). In the second case, the transaction is certified as if there is no optimistic delivery calling the code on Listing 5.4 (line 15). Afterwards, since there was a mismatch between the optimistic ant authoritative orders, the transactions in the `Certified` queue are put on the head of the `FSTDelivered` queue (lines 16-20) for being reprocessed.

The `processTerminated`, `commit` and `abort` procedures remain unchanged.

5.3.5 Termination Protocol with Partial Certification

In the termination protocol described in Section 5.2, the selection of which database sites receive each part of the read and write set results in being mandatory to use an atomic commitment protocol in order to ensure that all database sites agree on the transaction's outcome.

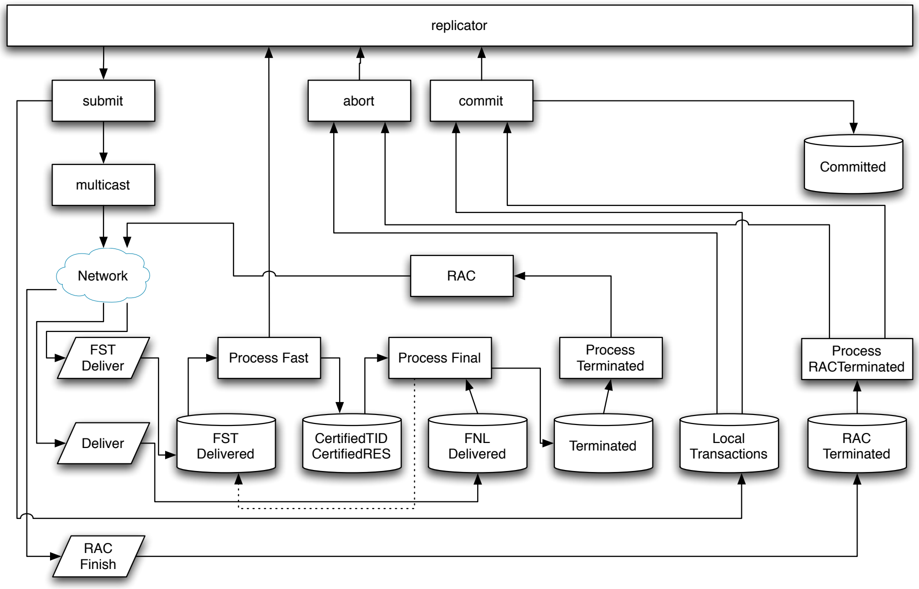


Figure 5.10: Termination protocol with partial certification.

Figure 5.10 describes the changes in the termination protocol in order to use an atomic commitment protocol. The use of an such a protocol imposes some restrictions on the certification of transactions in the `FNLDelivered` queue. A transaction at the head of the `FNLDelivered` queue can only be certified after the previous one finishes, i.e. after the previous one have been committed or aborted, as its outcome may influence the certification of the transaction at the head of the `FNLDelivered` queue. To fulfill this restriction the `processFinal` procedure is called only if there is no transaction waiting for the atomic commitment to finish.

Another change is that transactions in the `Terminated` queue can not be sent to the database, but instead the `processTerminated` procedure starts the atomic commit proposing the result of the certification as the outcome of the atomic commitment. This procedure is depicted in Listing 5.9. It starts by checking there is no atomic commitment running (line 2). Aftrewards it gathers information about the atomic commitment it is about to start (line 3), registers that an atomic commitment is running (line 4), and the identifier of the transaction associated to it (line 5). Finally, it starts the protocol by proposing its vote (line 6)

```
1  void processTerminated() {
     if(waiting_rac) return;
     CertWrapper<CertHeader> wtx=(CertWrapper<CertHeader>)Terminated.remove(0);
     waiting_rac=true;
5    waiting_tid=wtx;
     rac.propose(waiting_tid.getWrapped().tid(),wtx.isCommit());
   }
```

Listing 5.9: processTerminated procedure.

When an atomic commitment protocol finishes, the `RACTerminated` queue is used to store its outcome. The `processRACTerminated` procedure depicted in Listing 5.10 is called when the transaction it is waiting for is inserted into the `RACTerminated` queue.

```
1  void processRACTerminated() {
     boolean res=this.RAC_Terminated.remove(this.waiting_tid.getWrapped().tid());
     if(res)
       commit(waiting_tid.getWrapped());
5    else
       abort(waiting_tid.getWrapped());
     waiting_tid=null;
     waiting_rac=false;
   }
```

Listing 5.10: processRACTerminated procedure

The `processRACTerminated` procedure starts by removing the information associated to the transaction being finished from the `RACTerminated` queue (line 2). Afterwards depending on the outcome of the atomic commitment, it calls the `commit` (line 4) or `abort` (line 6) procedures. Finally it prepares enables the processing of the next transaction (lines 7-8), by eliminating references to the transaction waiting for the atomic commitment and setting that there is n atomic commitment running.

5.3.5.1 Resilient Atomic Commitment

As atomic commitment protocol, it is used the resilient atomic commitment protocol. The classes and interfaces used for its implementation are depicted in Figure 5.11.

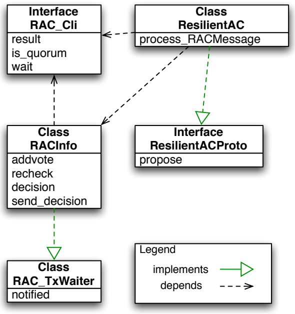


Figure 5.11: Classes and interfaces used in the resilient atomic commitment implementation.

The implementation has two main components, the `ResilientAC` class used for database sites start the resilient atomic commitment protocol and for message exchanging. The `RACInfo` class maintains the state of the protocol and notifies the protocol client when it needs some information. This information may be whether a set of votes is already sufficient to decide the protocol outcome, or to inform the database site that the protocol is expecting its vote.

The `ResilientAC` implementation relies on two procedures, `propose` which is used to multicast the database site vote for a transaction resilient atomic commitment, and the `processRACMessage` depicted in Listing 5.11, called when receiving a vote.

```
1  void processRACMessage(RACMessage m){
     RACInfo ac=this.ACs.get(m.id());
     if(ac==null){
       ac=new RACInfo(m.id(),members.length,cli);
5      this.ACs.put(m.id(),ac);
     }
     ac.addvote(m.vote(),m.from());
   }
```

Listing 5.11: processRACMessage procedure.

The first task of the `processRACMessage` procedure is to establish whether it is the first vote for the protocol (lines 2-3). If it is the first vote, then a new instance of `RACInfo` is created and inserted into the map of transaction identifiers to `RACInfo` (lines 4-5). Finally the vote is added to the instance of `RACInfo` (line 7).

A `RACInfo` instance represents a resilient atomic commitment. Its main tasks are, collect votes from other database sites, query its database site to establish which votes are sufficient for deciding commit, and to inform its database site of the protocol outcome. The vote collection is implemented by the `addvote` procedure depicted in Listing 5.12.

```
1   void addvote(boolean v,int from){
      vf[from]=true;
      if(!v){
        decided=true;
5       decision=false;
      }else{
        try{
          if(cli.is_quorum(rac_id,vf)){
            decided=true;
10          decision=true;
          }
        }catch(MissingTransactionException e){
          cli.wait(this,rac_id);
        }
15    }
      if(decided && !decision_sent){
        send_decision();
      }
    }
```

Listing 5.12: addvote procedure.

The `addvote` procedure starts by registering that database site `from` has voted for the protocol (line 2). Afterwards it analyses the vote, if it is an abort, then a decision is reached and the decision is to abort (lines 4-5). If the vote is to commit, then the client is queried to known whether the votes collected so far are sufficient to reach a decision (line 8). If the answer is positive then a decision is reached and the decision is to commit the transaction (lines 9-10). It is also possible that in the database site there is a lack of information about the transaction. In this case, when queried the database site throws a `MissingTransactionException` and the `RACInfo` instance reacts to it asking the database site to inform it when it knows about the transaction (line 13). Finally if during the processing of this vote a decision is reached the database site is informed of the decision (line 17).

5.4 Evaluation

This section describes the evaluation of the partial replication termination protocols proposed using the full replication termination protocol as the basis for comparison. All experiences were conducted in a centralized simulation model, allowing to combine simulated components with implementations of the atomic broadcast and termination protocols.

It starts with a description of the WAN topology used in the evaluation of the protocols, the application used and the clients setup. Afterwards we describe the simulation model and its configuration to reproduce the behavior of the real system. Finally, the results showing that partial replication requires fewer network and storage resources without sacrificing performance are presented.

5.4.1 Experimental Scenario

5.4.1.1 Network Topology and Database Sites Configuration

The network topology used to evaluate the termination protocols on the simulation infrastructure described consists on a WAN with 9 database sites. The database sites are distributed over 3 LAN connected to a central router in a star topology. Database sites in distinct LAN have 30ms latency and 4 communication hops between them.

5.4.1.2 Application Profile

The application used for evaluating the replication protocols is the TPC-C industry standard benchmark [TPC01]. It is worth noting that, the interest in TPC-C to evaluate the replication protocols is just in the workload specified by this benchmark. Therefore, the benchmark constraints of throughput, performance, screen load and background execution of transactions are not considered. As the model is coarse grained (e.g. the cache is modeled by a hit ratio), it is also not necessary to observe the requirement to discard the initial 15 minutes of each run. The content of each request is generated according to a simulated user based on the TPC-C benchmark [TPC01].

The TPC-C benchmark proposes a wholesale supplier with a number of geographically distributed sales districts and associated warehouses as an application. This environment simulates an OLTP workload with a mixture of read-only and update intensive transactions. A client can request five different transactions: *New Order*, adding a new order into the system (with 44% probability of occurrence); *Payment*, updating the customer's balance, district and warehouse statistics (44%); *Order Status*, returning a given customer latest order (4%); *Delivery*, recording the delivery of products (4%); and *Stock Level*, determining the number of recently sold items that have a stock level below a specified threshold (4%). Database contents and transaction mix are summarized in Tables 5.1 and 5.2.

From the transaction mix, *delivery* transactions are CPU bound; *payment* transactions are prone to conflicts by updating a small number of data items in the

	Number of items		
Relations	100 (Cli.)	1000 (Cli.)	**Tuple size**
Warehouse	10	100	89 bytes
District	1×10^2	1×10^3	95 bytes
Customer	3×10^5	3×10^6	655 bytes
History	3×10^5	3×10^6	46 bytes
Order	3×10^5	3×10^6	24 bytes
New Order	9×10^4	9×10^5	8 bytes
Order Line	3×10^6	3×10^7	54 bytes
Stock	1×10^6	1×10^7	306 bytes
Item	1×10^5	1×10^5	82 bytes
Total	$\approx 5 \times 10^6$	$\approx 5 \times 10^7$	

Table 5.1: Size of tables in TPC-C.

Transaction	**Probability**	**Description**	**Read-only?**
New Order	44%	Adds a new order into the system.	No
Payment	44%	Updates the customer's balance, district and warehouse statistics.	No
Order Status	4%	Returns a given customer's latest order.	Yes
Delivery	4%	Records the delivery of orders.	No
Stock Level	4%	Determines the number of recently sold items that have a stock level below a specified threshold.	Yes

Table 5.2: Transaction types in TPC-C.

Warehouse table; *payment* and *order status* execute some code conditionally. The database size is configured for each simulation run according to the number of clients as each warehouse supports 10 emulated clients [TPC01]. As an example, with 100 clients, the database contains in excess of 5×10^6 tuples, each ranging from 8 to 655 bytes.

In partial replication setups, database relations history, order, new order and order line have been horizontally fragmented by warehouse. The other relations, warehouse, district, customer, stock and item have not been fragmented and are replicated by all database sites.

5.4.1.3 Client

A database client is attached to a database site and produces a stream of transaction requests. After each request is issued, the client blocks until the server replies, thus modeling a single threaded client process. After receiving a reply, the client is then paused for some amount of time (think-time) before issuing the next transaction request. The content of each request is generated according to a simulated user based on the TPC-C benchmark [TPC01].

During the run of the simulation, the client logs the time at which a transaction is submitted, the time at which it terminates, the outcome (either abort or commit) and a transaction identifier. The latency, throughput and abort rate of the server can then be computed for one or multiple users, and for all or just a subclass of the transactions.

5.4.2 Simulation Model

The simulation model proposed uses the centralized simulation model of [AC97]. It allows to combine simulated environment components, such as database servers, database clients and network, using discrete-event simulation, with real code for the components under study, namely the atomic broadcast and the termination protocols.

Figure 5.12, describes the architecture of the simulation model, and the following sections describe its components:

The simulation kernel offering the primitives upon which the simulation model is implemented ;

The centralized simulation runtime (CSRT) responsible for the integration of simulated code with real code;

The database server responsible for local database management;

The database client responsible for the workload generation, i.e. submitting transactions to database servers;

The network responsible for carrying messages exchanged by database servers.

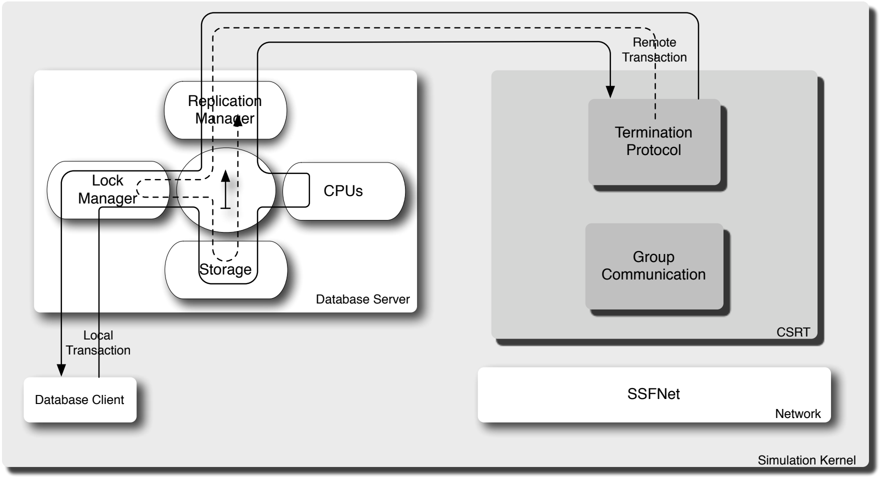


Figure 5.12: Architecture of the simulation model.

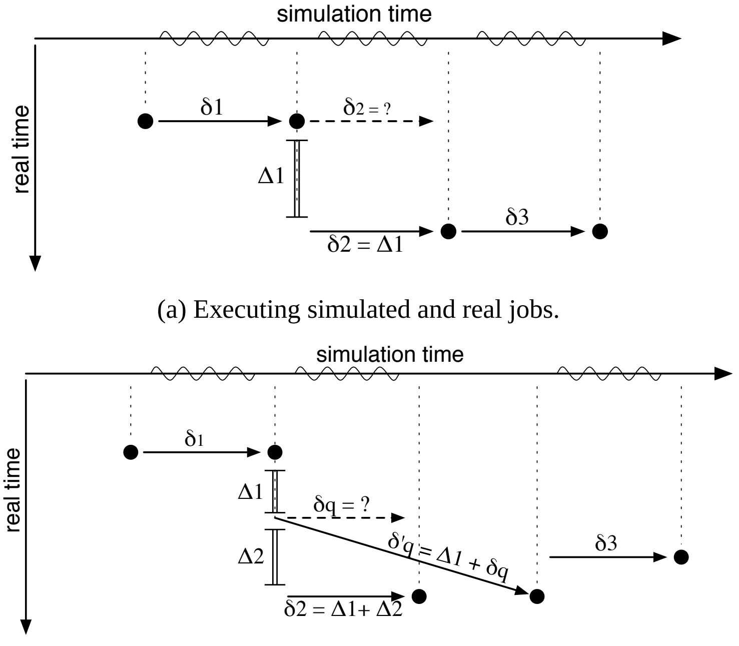
5.4.2.1 Simulation Kernel

SSF and SSFNet The simulation kernel is based on the Java version of the Scalable Simulation Framework (SSF) [Cow99], which provides a simple yet effective infra-structure for discrete-event simulation [SSF04]. It comprises five base interfaces: *Entity*, *Process*, *Event*, *inChannel* and *outChannel*. An entity owns processes and channels, and holds simulation state. The communication among entities in the simulation model occurs by exchanging events through channels: An event is written to an *outChannel*, which relays it to all connected *inChannels*. Processes retrieve events by polling. Simulation time is updated according to delays associated with event transmission. The simulation kernel includes also a simple language, that can be used to configure large models from components by instantiating concrete entities and connecting them with channels.

Simulation models are therefore built as libraries of components that can be reused. This is the case of the SSFNet framework [CNO99], which models network components (e.g. network interface cards and links), operating system components (e.g. protocol stacks), and applications (e.g. traffic generators). Complex network models can be configured using such components, mimicking existing networks or exploring particularly large or interesting topologies. The SSFNet framework provides also extensive facilities to log events.

Centralized Simulation A centralized simulation model combines real software components with simulated hardware, software and environment components to model a distributed system. It has been shown that such models can accurately reproduce the performance and dependability characteristics of real systems [AC97]. The centralized nature of the system allows for global observation of distributed computations with minimal intrusion as well as for control and manipulation of the experiment.

The execution of the real software components is timed with a profiling timer and the result is used to mark the simulated CPU busy during the corresponding period, thus preventing other jobs, real or simulated, to be attributed simultaneously to the same CPU. In detail, a simulated CPU is obtained as follows: A boolean variable indicates whether the CPU is busy and a queue holds pending jobs, with their respective durations. A job with duration δ can be executed at a specific instant t by scheduling a simulation event to enqueue it at simulated time t. If the CPU is free, the job is dequeued immediately and the CPU marked as busy. A simulation event is then scheduled with delay δ to free the CPU. Further pending jobs are then considered.



(a) Executing simulated and real jobs.

(b) Scheduling events from real code.

Figure 5.13: Handling simulated and real jobs in centralized simulation.

Executing jobs with real code is layered on top of the same simulation mechanism. Figure 5.13 illustrates this with an example of how three queued jobs are executed. The second job is assumed to contain real code. The x-axis depicts simulated time and the y-axis depicts relevant real-time (i.e. real-time consumed during execution of pure simulation code is ignored and thus pure simulation progresses horizontally). The x-axis shows also with a wiggly line when the simulated CPU is busy. Solid dots represent the execution of discrete simulation events. Scheduling of events is depicted as an arrow and execution of real code as a double line.

The first job in the queue is a simulated job with duration δ_1. The CPU is marked as busy and an event is scheduled to free the CPU. After δ_1 has elapsed, execution proceeds to a real job. In contrast with a simulated job, it is not known beforehand which is the duration δ_2 to be assigned to this job. Instead, a profiling timer is started and the real code is run. When it terminates, the elapsed time Δ_1 is measured. Then $\delta_2 = \Delta_1$ is used to schedule a simulation event to proceed to the next job. This brings into the simulation time-line the elapsed time spent in a real computation. Finally the second simulated job is run with duration δ_3 and the CPU marked as free afterward, as the queue is empty.

As a consequence of such setup, queuing (real code or simulated) jobs from simulated jobs poses no problem. Only when being run, they have to be recognized and treated accordingly. Problems arise only when real code needs to schedule simulation events, for instance, to enqueue jobs at a later time. Consider in Figure 5.13b a modification of the previous example in which the third job is queued by the real code with a delay δ_q. If real code is allowed to call directly into the simulation runtime two problems would occur:

- Current simulation time still does not account for Δ_1 and thus the event would be scheduled too early. Actually, if $\delta_q < \Delta_1$ the event would be scheduled in the simulation past!

- The final elapsed real time would include the time spent in simulation code scheduling the event, thus introducing an arbitrary overhead in δ_2.

These problems can be avoided by stopping the real-time clock when re-entering the simulation runtime from real code and adding Δ_1 to δ_q to schedule the event with a delay δ'_q. The clock is restarted upon returning to real code and thus δ_2 is accurately computed as $\Delta_1 + \Delta_2$. In addition to safe scheduling of events from simulation code, which can be used to communicate with simulated network and application components, the same technique must be used to allow real code to read the current time and measure elapsed durations.

5.4.2.2 Replicated Database Model

The replicated database is modeled as a set of database sites. Each database site is modeled as a stack of components, configured as a hosts in a SSFNet network, and includes a number of clients issuing transaction requests. After being executed and when entering the committing state, transactions are submitted to the termination protocol, which uses group communication to disseminate the updates to other replicas. When finished, the outcome of the termination protocol is returned to the client.

Database Server The database server is modeled as a scheduler and a collection of resources, such as storage and CPUs, and a concurrency control policy (Lock Manager). A database server handles multiple clients, executing transactions submitted by them. Each transaction is modeled as a sequence of operations, which can be one of: *i*) fetch a data item; *ii*) do some processing; *iii*) write back a data item. Upon receiving a transaction request each operation is scheduled to execute on the corresponding resource. The transaction execution path is shown on Figure 5.12 as a solid line traversing the resources of the database server. The processing time of each operation is previously obtained by profiling a real database server.

First, operations fetching and storing items are submitted to the Lock Manager for concurrency control. Depending on the policy being used, the execution of a transaction can be blocked between operations. When a transaction commits, all other transactions waiting on locks of written items are aborted due to write-write conflicts. If the transaction aborts, the locks are released and can be acquired by the next transaction. In addition, all locks are atomically acquired when the transaction starts executing, and atomically released when the transaction commits or aborts, thus avoiding the need to simulate deadlock detection. This is possible as all items accessed by the transaction are known beforehand.

Processing operations are scaled according to the configured CPU speed. Each is then executed in a round-robin fashion by any of the configured CPUs. Notice that a simulated CPU accounts both simulated jobs and real jobs scheduled using the centralized simulation runtime. Real jobs have higher priority, so a simulated transaction executing can be preempted to reassign the simulated CPU to a real job.

A storage element is used for fetching and storing data items and is defined by its latency and number of allowed concurrent requests. Each request manipulates a single storage sector, hence storage bandwidth becomes configured indirectly. A cache hit ratio determines the probability of a read request being handled instantaneously without consuming storage resources.

When a commit operation is reached, the corresponding transaction enters the termination protocol. This involves the identification of data items read and written as well as the values of the written data items. As the termination protocol is handled by real code, the representation of data item identifiers and values of updated data items must accurately correspond to those of real traffic. When the termination protocol is concluded, the transaction is committed by finishing writing and releasing all locks held. The outcome can then be returned to the issuing client. Remotely initiated transactions must also be handled. In this case, locks are acquired before writing to disk. However, as such transactions have already successfully concluded the termination protocol and must be committed, local transactions holding the same locks are preempted and aborted right away. Note that local transactions which conflict with concurrent committed transactions would abort later during certification anyway.

5.4.3 Model Instantiation

This section describes how the model is configured to reproduce the behavior of a real system. This allows to validate the model by comparing the results of simple benchmarks run both in the simulator and in a real system.

5.4.3.1 Configuration Parameters

The model is configured according to the equipment used for testing. In this case, a dual processor AMD Opteron at 2.4GHz with 4GB of memory, running the Linux Fedora Core 3 Distribution with kernel version 2.6.10. For storage it uses a fiber-channel attached box with 4, 36GB SCSI disks in a RAID-5 configuration and the Ext3 file system.

The configuration of the centralized simulation runtime reduces to four parameters: fixed and variable CPU overhead when a message is sent and received. These can easily be determined with a simple network flooding benchmark with variable message sizes (described in detail in Section 5.4.3.2).

The database server configuration issues are the CPU time and disk bandwidth consumed by each transaction. The amount of CPU consumed by the execution of each transaction is tightly related to the database system used and to the size of the database, although not significantly affected by concurrency. In order to obtain these parameters, we use a profiled PostgreSQL [PSQ] running the TPC-C benchmark [TPC01], configured for up to 2000 clients but with a small number of actual running clients. In PostgreSQL, each process handles a single transaction from start to end, so this configuration task reduces to profiling a process in the host operating system.

In detail, the task of profiling a process uses the CPU time-stamp counter which provides accurate measure of elapsed clock cycles. By using a virtualization of the counter for each process [Pet04], it is also possible to obtain the time elapsed only when the process is scheduled to run. A comparison of these timers is used to estimate the time that the process is blocked, most likely waiting for I/O. To minimize the influence in the results, the elapsed times are transmitted over the network only after the end of each query (and thus out of the measured interval), along with the text of the query itself.

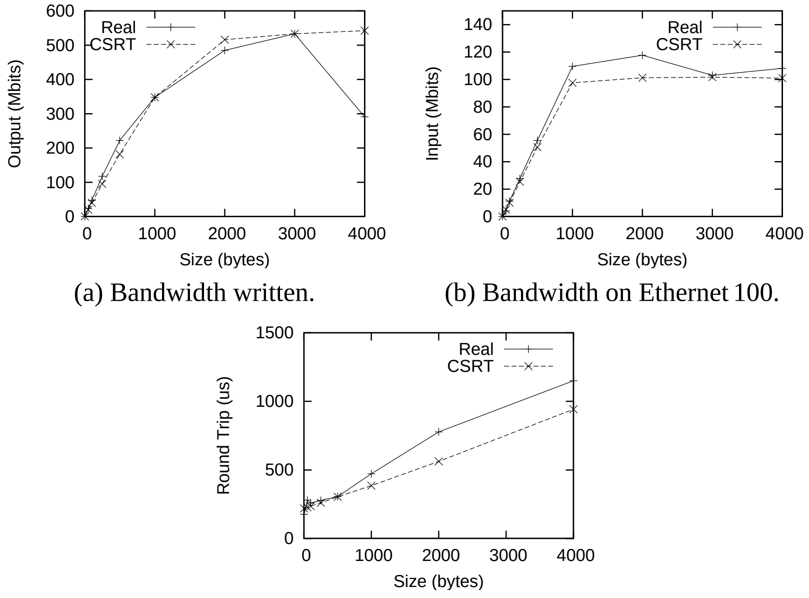
The time consumed by the transaction execution is then computed from the logs. By examining the query itself, each transaction is classified. Interestingly, the processor time consumed during commit is almost the same for all transactions (i.e., less than 2ms). In read-only transactions the real time of the commit operation equals processing time, meaning that no I/O is performed. This does not happen in transactions that update the database. The observation that the amount of I/O during processing is negligible indicates that the database is correctly configured and has a small number of cache misses.

After discarding aborted transactions and the initial 15 minutes, as specified by the TPC-C standard, the system runs 5000 transactions and uses the resulting logs to obtain empirical distributions for each transaction class. Some transaction classes perform some work conditionally and would produce bimodal distributions. However, as analysis of results is simplified if each transaction class is homogeneous, these transactions were split in two different classes.

Throughput for the storage was determined by running the IOzone disk benchmark [IOZ04] on the target system with synchronous writes of 4KB pages and a variable number of concurrent process. This resulted in a maximum throughput of 9.486MBps. As the cache hit ratio observed has always been above 98%, the simulation was configured with a hit ratio of 100%. This means that read data items do not directly consume storage bandwidth. CPU resources are already accounted for in the CPU times as profiled in PostgreSQL.

5.4.3.2 Validation

The model and its configuration were validated by comparing the resulting performance measurements of the model to those of the real system running the same benchmark. Notice that abstraction prevents the model to completely reproduce the real system and thus the validation step is only approximate. This is acceptable as simulated components are used only to generate a realistic load and not the subject themselves of evaluation. The validation of the SSFNet has been done previously [CNO99].



(a) Bandwidth written.

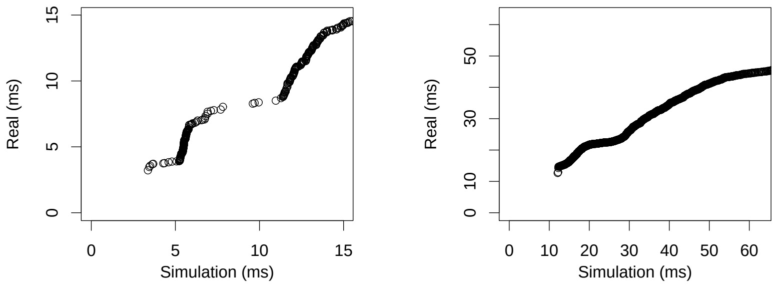
(b) Bandwidth on Ethernet 100.

(c) Averageround-trip.

Figure 5.14: Validation of the centralized simulation runtime.

The centralized simulation kernel is configured and validated according to [AC97]. Figure 5.14a shows the maximum bandwidth that can be written to an UDP socket by a single process in the test system with various message sizes. Notice that crossing the 4KB virtual memory page boundary impacts performance in the real system. This is most likely due to memory management overhead, but it is irrelevant for the presented results as the protocol prototype uses a smaller maximum packet size. Figure 5.14b shows the result of the same benchmark at the receiver, limited by the network bandwidth. Finally, Figure 5.14c shows the result of a round-trip benchmark. The difference observed with packets with size greater than 1000 bytes is due to SSFNet not enforcing the Ethernet MTU in UDP/IP traffic. Deviations from the real system are avoided by restricting the size of packets used to a safe value.

To validate the architecture of the simulated database server which has been used before in [ACL87], a run of the TPC-C benchmark with only 20 clients and a total of 5000 transactions was used. This solution was adopted as it is not feasible to run the benchmark in a real setting with thousands of clients. Quantile-quantile plots (Q-Q plot) of observed latencies is presented in Figure 5.15 showing that simulation results successfully approximate the real system.



(a) Read-only transactions.

(b) Update transactions.

Figure 5.15: Validation of transactions latency.

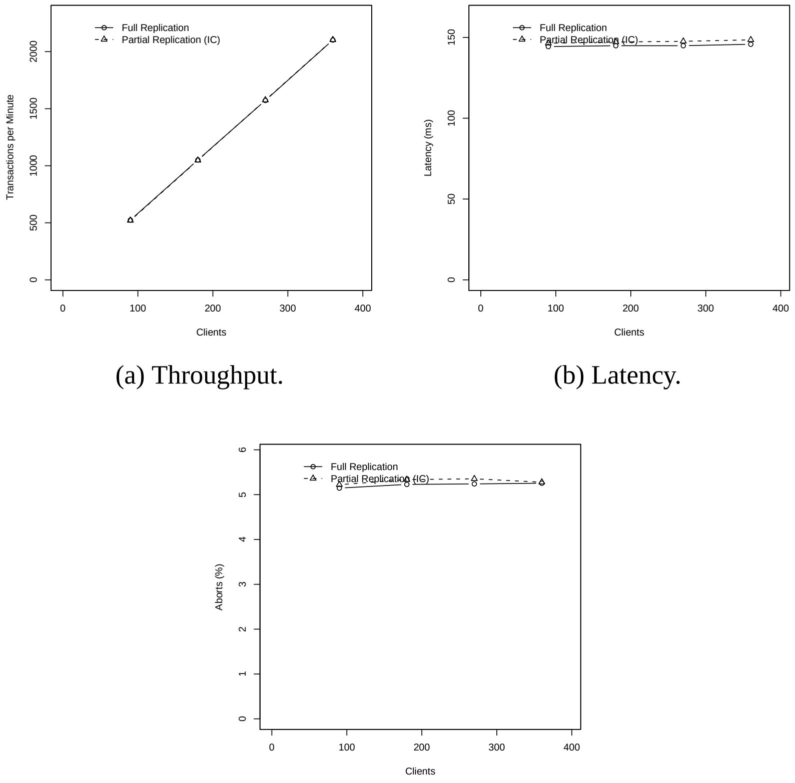
5.4.4 Experimental Results

Partial replication has been proposed as a technique to reduce resource requirements while ensuring performance comparable to full replication. In this section demonstrates that partial replication termination protocols fulfill the requirements of requiring less network bandwidth and local storage while providing a performance similar to the full replication termination protocol, in terms of transaction throughput and abort rate.

To demonstrate our goal we start by comparing the full replication termination protocol with the partial replication termination protocol with independent certification. As you may recall from Section 5.2.2.2, these protocols are similar, apart from the fact that the partial replication termination protocol only sends the write values to the replicas of the referred data items.

Figure 5.16 presents the protocols performance in terms of transaction throughput, latency and abort rate. The values presented consider a variable number of clients in the 90 to 360 clients interval. From the Figure 5.16, it can be observed that the protocols present similar values of transaction throughput, execution latency and abort rate. This is an expected result due to the similarity of both protocols, and a positive result for partial replication as it presents performance comparable to full replication, accomplishing one of the proposed objectives of partial replication.

The other objective of partial replication, and the motivation to its adoption instead of full replication, is that partial replication should required fewer resources at the local storage and network bandwidth. The demonstration that partial replication also fulfills these requirements is presented in the following. Regarding storage requirements its bandwidth utilization is presented in Figure 5.17a, and, as



(a) Throughput.

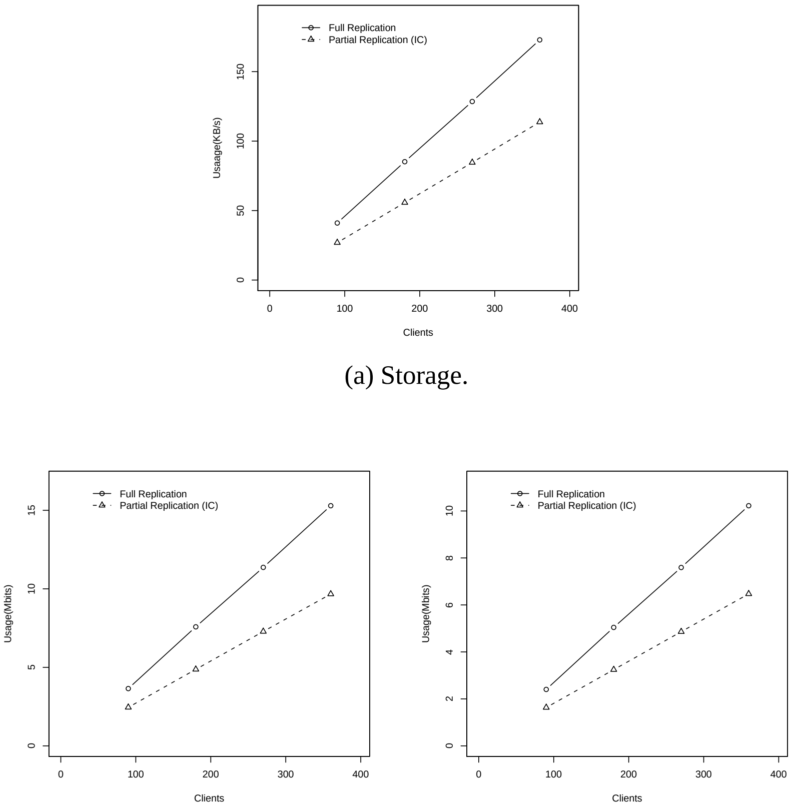
(b) Latency.

(c) Abort rate.

Figure 5.16: Protocols performance.

expected, partial replication effectively reduces the required storage bandwidth. Furthermore, it can be observed that the growth of the required storage bandwidth with the number of clients is smaller for partial replication than for full replication. The smaller growth in the storage bandwidth requirements for partial replication results from the partitioning used. For the tables partially replicated, there is a linear growth of its size with the number of clients for full replication, while this growth is only of one third in the case of partial replication. This partitioning reduces the write load when compared to full replication, resulting also in a smaller growth on the required storage bandwidth with an increase in the number of clients, and consequent database growth.

Regarding the required network bandwidth, its utilization at the central router and at the LAN router is presented in Figures 5.17b and 5.17c respectively. It can be observed that the full replication termination protocol requires more bandwidth



(a) Storage.

(b) Central Router.

(c) LAN router.

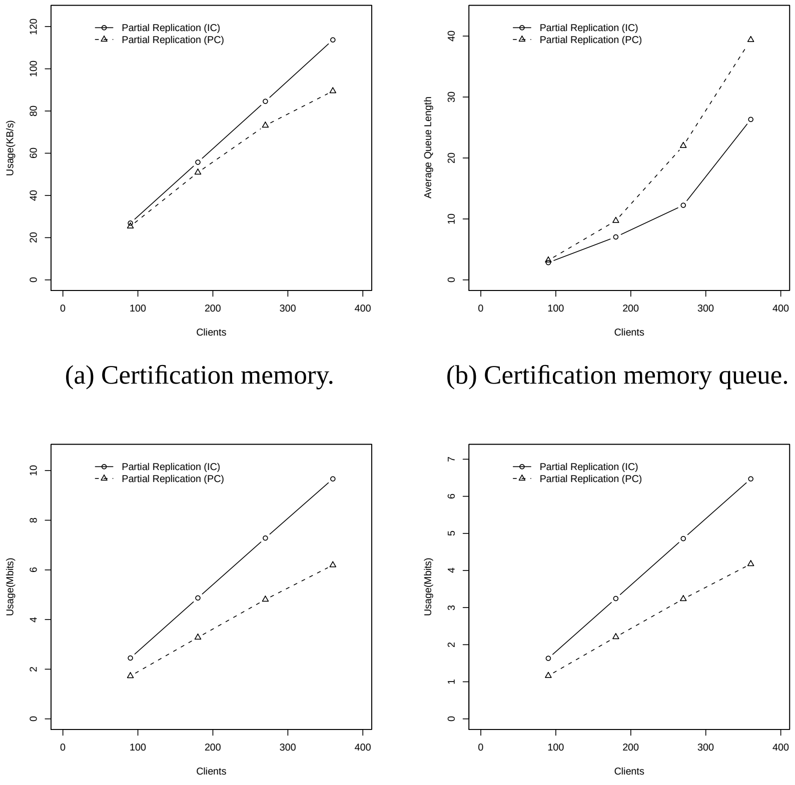
Figure 5.17: Resources usage.

than the partial replication termination protocol with independent certification. At the central router the required bandwidth is also higher than at the LAN router, as it must handle the network traffic generated at each LAN target to the other two LAN.

At this point our goal is accomplished. Partial replication with independent certification reduces the required storage and network bandwidths while maintaining the overall performance. The goal now is to establish if the partial replication with partial certification termination protocol can reduce even further the required network bandwidth for the partial replication with independent certification termination protocol.

The partial replication with independent certification termination protocol requires the transmission of the full read and write sets over the network, which may require more network bandwidth than the required for the RAC protocol used by the partial replication with partial certification termination protocol.

In the following, for both partial replication termination protocols the memory requirements for storing the information used for certify concurrent transactions is depicted in Figure 5.18a and the number of transactions used for certification is depicted in Figure 5.18b. The network bandwidth used at the central router and at the LAN router are depicted in Figure 5.18c and in Figure 5.18d respectively. In terms of storage there are no differences in the written data by both protocols, resulting in the same storage usage which have been presented in Figure 5.17a.



(a) Certification memory.

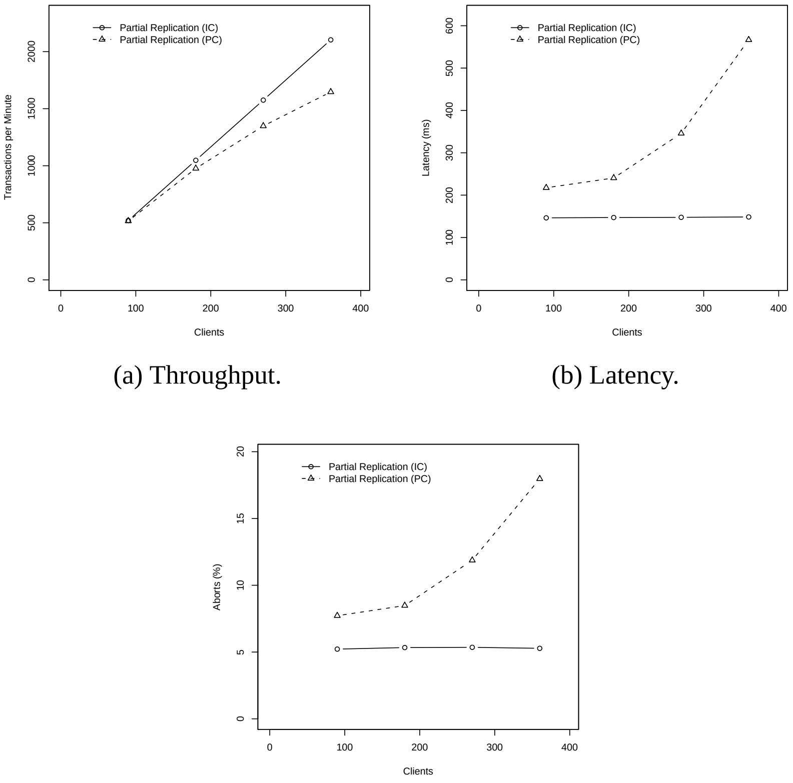
(b) Certification memory queue.

(c) Central Router.

(d) LAN router.

Figure 5.18: Resources usage.

The memory utilization as well as the number of transactions used for certification, are quite low in both cases. This excludes the memory utilization as a parameter influencing the decision on the best protocol for the selected scenarios.



(a) Throughput.

(b) Latency.

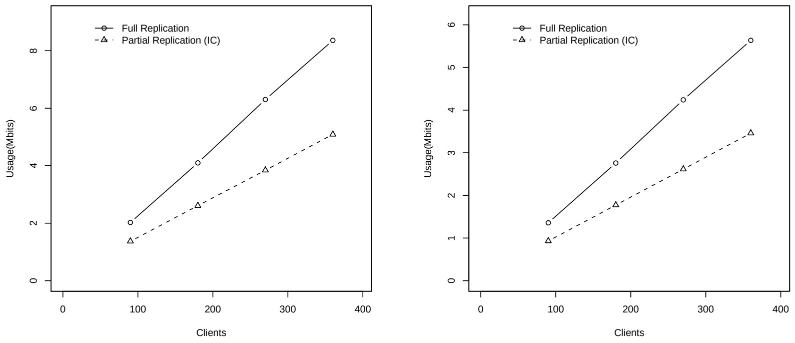
(c) Abort rate.

Figure 5.19: Protocols performance.

From the network bandwidth utilization figures it can be seen that the termination protocol with partial certification (PC) requires less network bandwidth than the termination protocol with independent certification (IC), i.e., the network bandwidth required for the RAC protocol is smaller than the network bandwidth required to transmit the full read and write sets. The divergence of the network bandwidth requirements with the increase in the number of clients, is a consequence of the lower performance values presented by the termination protocol with partial certification, as can be observed in Figure 5.19

From Figure 5.19, it can be observed a decrease in the performance of the termination protocol with partial certification with the increase in the number of clients, when compared with the termination protocol with independent certification. This decrease is a consequence of the latency growth which can be observed in Figure 5.19b and of the growth of the abort rate which can be observed in Figure 5.19c.

As a conclusion of the analysis of the partial replication termination protocols, it can be said that, the usage of the termination protocol with partial certification is preferable to the termination protocol with independent certification in terms of network bandwidth as it requires fewer resources. In terms of performance, above 180 clients, the termination protocol with partial certification presents lower transaction throughput, and much higher latencies and abort rates, which may advise the utilization of the termination protocol with independent certification in those situations, and the termination protocol with partial certification until 180 clients.



(a) Central Router.

(b) LAN router.

Figure 5.20: Network bandwidth usage.

Regarding resource usage, the termination protocols using the serializable consistency criterion require the transmission of the read and write sets over the network. On the other hand, if using snapshot isolation as the consistency criteria, then only the write sets need to be transmitted over the network. This should narrow the difference in the network bandwidth required by the full and partial replication termination protocols, and also between the two partial replication termination protocols.

As the partial replication with independent certification termination protocol generally presents better results than the partial replication with partial certification termination protocol, it will be the selected one for the comparison of the termination protocols using snapshot isolation as the consistency criterion. The network bandwidth utilization graphics are presented in Figure 5.20 and, as expected, present a reduction in the required network bandwidth, compared to the one presented in Figure 5.18 for the same termination protocols using the serializable consistency criterion.

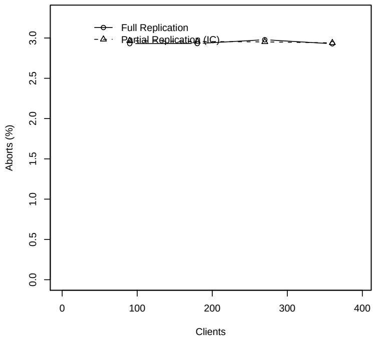


Figure 5.21: Snapshot isolation abort rate.

In terms of performance, there are no differences in the transactions throughput and execution latency from the values presented in Figure 5.16. In terms of abort rate, the utilization of the snapshot isolation consistency criterion reduces the abort rate, as can be seen by observing the values of the abort rate for the serializable consistency criteria presented in Figure 5.16c and those for the snapshot isolation consistency criterion presented in Figure 5.21.

5.5 Summary

This chapter described the major contributions of this thesis. The partial replication termination protocols and the simulation environment used to evaluate the termination protocols.

This chapter started with a review of database replication with optimistic execution, presenting the database model and a detailed description of its components.

Following that review we introduced partial replication and the changes it requires to the model presented earlier. The changes occur in the transaction and termination managers.

Regarding the transaction manager, two possibilities should be considered. Either the database site executing a transaction has a replica of every data item accessed by the transaction or not. The first case, requires no changes in the transaction manager, and should be the common situation in a well fragmented database. The second case requires changes in the transaction manager in order to access the database sites holding a replica of the data required by the transaction, and has been the subject of the work in [Jún04].

The major differences between partial and full replication occur in the termination manager. To deal with partial replication issues we proposed two approaches. One uses a protocol similar to the full replication termination protocol, in which every database site receives the complete read and write sets and independently certify every transaction. The second approach is to send to each database site only the relevant part of the read and write sets. In this case, each database site only certifies part of the transaction and an atomic commitment protocol is required in order to establish the transaction outcome.

In the proposed partial replication termination protocols, transactions are ordered before being certified, which results in all database sites reaching the same decision on the conflicting data items. To take advantage of all replicas of a data item reaching the same decision regarding its conflicts, we propose a new definition for the atomic commitment problem – the resilient atomic commit (RAC), which allows to decide to commit transactions as long as at least a replica of each data item accessed by the transaction does not fail.

The JAVA implementation of the termination protocols was described in Section 5.3 and its evaluation in Section 5.4.

To evaluate the termination protocols we proposed a simulation model using the centralized simulation model which allows to combine the implementations of the termination protocols with with simulated components such as the network, the database server and database client.

The TPC-C benchmark as been used to evaluate the termination protocols. From the experiences conducted, it was possible to demonstrate that partial replication effectively reduces the storage and network bandwidth requirements. It also shows that up to 180 clients partial replication with partial certification presents higher savings in network bandwidth, while maintaining similar performance to the full and partial replication with independent certification termination protocols. Above 180 clients the latency and abort rate of partial replication with partial certification grows rapidly, making the use of the partial replication with independent certification termination protocol preferable.

In cases where the application tolerates a consistency criterion weaker than the serializable consistency criterion, then the use of the snapshot isolation consistency criterion may be a solution to reduce the network bandwidth requirements. In this case only the write sets need to be transmitted over the network thus reducing the required network bandwidth.

Chapter 6

Conclusions

The replication and communication protocols behave differently depending on the network topology, i.e., their behavior changes depending on being executed in a LAN or in a WAN. The responsible for the changes in the protocols behavior is the increased latency, and point to point latency variations, presented in WAN scenarios.

The WAN increased latency reduces the network spontaneity used by the optimistic total order protocols, rendering its adoption questionable in such a scenario. In the replication protocols, the increase in the latency augments the number of concurrent transactions augmenting also the probability of conflicts between transactions, and the abort rate.

Regarding the optimistic total order protocols, in a LAN, a message sent to several destinations is received in the same order by all destinations with high probability. Optimistic total order protocols take advantage of this property of the LAN to tentatively deliver messages, assuming their tentative order matches the authoritative order, allowing for earlier processing of the messages.

In a WAN the network spontaneity is much lower, but can be improved using the strategy proposed by the statistically estimated optimistic total order protocol. Using this protocol, it is possible, in some network configurations, to improve the network spontaneity to values above 80% with message rates up to 250 messages per second. Unfortunately, it is not possible to obtain such a level of network spontaneity for all message rates. Above certain message rates, the network spontaneity degrades, becoming similar to the network spontaneity without using the strategy of the statistically estimated optimistic total order protocol.

Some interesting properties of the statistically estimated optimistic total order protocol, are that the higher values of spontaneity are obtained without sacrificing the optimistic window which is only marginally inferior to the one of the sequencer based protocol. The protocol latency is only marginally augmented, due to the adjusts made in the sequencer that delays the ordering of its own messages, reflecting these delays into the protocol latency.

The effectiveness of the approach and validity of the results obtained by the statistically estimated optimistic total order protocol have been independently confirmed in an evaluation of total order protocols in wide area networks [ADGS03], and has been the basis of some other works on optimistic total order [RMC06, MRR06]. The protocol have been also implemented in the Appia toolkit [MPR01].

Regarding the replication protocol, with partial replication it is possible to improve system dependability while reducing the required network bandwidth and storage space. These results are obtained by judiciously selecting which data is replicated at each replica, avoiding to replicate remotely, data that will not be used by the remote replicas.

The replication protocols with optimistic execution minimize replicas interaction by locally executing transactions and only interacting with the other replicas at commit time. The replicas interaction is governed by the termination protocol which can be decomposed in three generic steps, transaction's dissemination and ordering, transaction's certification and transaction's outcome agreement.

In the first step, transaction information is disseminated to all replicas. The transaction information can be split in two sets, the concurrency control information and the produced data. The information disseminated during the first step dictates the available choices for the certification and agreement steps.

Regarding the produced data, it is only of interest for the replicas replicating it, so there are no advantages in sending it to every replica, which have to discard part of the produced data before writing the data to stable storage. If the produced data is sent to every replica, then the network bandwidth used will be wasted and the objective of reducing its usage compromised.

The partial replication with partial certification termination protocol sends to each replica only the parts of the read and write sets related to the data they replicate. This allows to reduce the network bandwidth utilization, although requiring the execution of the outcome agreement step, responsible for some network bandwidth utilization. As the agreement step we propose the resilient atomic commit. It presents lower latency than the existing atomic commitment protocols, but still requires a communication step being penalized by the network latency. For a small number of clients the effects of this additional communication step is not reflected in the performance values, it is only visible in the transaction's execution latency.

With the growth of the number of clients its effects start to be noticeable also in the performance values, rendering its utilization questionable in those scenarios.

The partial replication with independent certification termination protocol, sends the read and write sets to every replica independently of which data it replicates. Comparing to the previous approach it increases the network bandwidth utilization, but does not require the execution of the outcome agreement step. Instead it uses the certification test result to impose the transaction outcome. As a result of not requiring the additional communication step it presents lower transaction execution latencies and is able to support a higher number of clients without degrading its performance as happens with the partial replication with partial certification termination protocol. It is the right choice for a larger number of clients, as it presents a reduction on the required network bandwidth when compared to full replication, while maintaining the same performance.

The memory required to hold the transaction information of committed transactions used in the certification of concurrent transactions has been defined as a comparison parameter among termination protocols. It results from the values obtained, that this parameter is not crucial as the memory used to hold this kind of information is always low, and the memory used by each protocol is almost the same. The per transaction memory for the partial replication with partial certification termination protocol is inferior to the one required by the partial replication with independent certification termination protocol. However, the number of transactions being stored to be used by the certification is higher due to the higher latency of the partial replication with partial certification termination protocol. This results the memory used by both protocols being almost the same.

The network bandwidth used by the partial replication protocol can be further reduced by changing the adopted consistency criterion from serializable to snapshot isolation. Databases using snapshot isolation as the consistency criterion are gaining popularity because of the performance improvements that may be achieved relatively to serializable databases. With the proper precautions it is possible that snapshot isolation executions are serializable [FLO$^+$05]. In this case the network bandwidth savings result from the fact that for snapshot isolation certification only the write sets are required.

6.1 Future Work

The statistically estimated total order protocol improves the spontaneity in a WAN, allowing the use of optimistic protocols in such networks. These protocols are interesting as their optimistic delivery may be used to mask some of the WAN latency, if it allows the early start of other tasks. For instance, optimistic delivery allow for optimistic certification of transactions, and to start to update the database immediately after the reception of the transaction authoritative delivery, when the optimistic and authoritative orders match. Another improvement that should be possible, when using optimistic certification, is to optimistically start the database update, allowing the authoritative delivery to immediately commit the transaction.

The dissemination protocol uses point to point connections to send the transaction information to all replicas. In a LAN where there are no bandwidth limitations and the point to point latencies among any two database sites is the same, this seems to be the obvious choice. In a WAN, usually not all database sites have the same bandwidth and their point to point latency is also different. If the dissemination protocols can take advantage of the network topology to reduce the required bandwidth at the database sites and possibly also reducing the overall latency, then the replication protocols may benefit from such improvements.

Bibliography

[AAAS97] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'97)*, volume 1300 of *Lecture Notes in Computer Science*, Passau (Germany), August 1997. Springer.

[AAC91] M. Ahamad, M. Ammar, and S. Cheung. Multidimensional voting. *ACM Transactions on Computer Systems*, 9(4):399–431, 1991.

[AC97] G. Alvarez and F. Cristian. Applying simulation to the design and performance evaluation of fault-tolerant systems. In *Proceedings of the* 16^{th}*IEEE International Symposium on Reliable Distributed Systems*. IEEE Computer Society Press, October 1997.

[ACBMT95] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Technical Report TR95-1534, Department of Computer Science, Cornell University, November 1995.

[ACL87] R. Agrawal, M. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *ACM Transactions on Database Systems*, 12(4), 1987.

[ACM95] G. Alvarez, F. Cristian, and S. Mishra. On-demand asynchronous atomic broadcast. In *Proc. the* 5^{th} *IFIP Working Conf. Dependable Computing and Critical Applications*, Urbana-Champaign, IL, September 1995.

[ADGS03] T. Anker, D. Dolev, G. Greenman, and I. Shnayderman. Evaluating total order algorithms in wan. In *International Workshop on Large-Scale Group Communication, held in conjunction with the* 22^{nd} *Symposium on Reliable Distributed Systems (SRDS)*, October 2003.

[ADKM92a] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *WDAG '92: Proceedings of the* 6^{th} *International Workshop on Distributed Algorithms*, volume 647 of *Lecture Notes in Computer Science*, pages 292–312, London, UK, November 1992. Springer.

[ADKM92b] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *Proceedings of the* 22^{nd}*IEEE International Symposium on Fault-Tolerant Computing*, pages 76–84. IEEE Computer Society Press, July 1992.

[ADMA$^+$02] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. Practical wide-area database replication. Technical Report CNDS-2002-1, Johns Hopkins University, 2002.

[AGP02] M. Abdallah, R. Guerraoui, and P. Pucheral. Dictatorial transaction processing: Atomic commitment without veto right. *Distributed and Parallel Databases*, 11(3):239–268, 2002.

[Alo97] G. Alonso. Partial database replication and group communication primitives (extended abstract). In *Proceedings of the* 2^{nd} *European Research Seminar on Advances in Distributed Systems (ERSADS'97)*, pages 171–176, Zinal (Valais, Switzerland), 1997.

[AMMS$^+$95a] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. The totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, 1995.

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

[Anc96] E. Anceaume. A lightweight solution to uniform atomic broadcast for asynchronous systems: proofs. Technical Report PI-1066, IRISA, November 1996.

[AT02] Y. Amir and C. Tutu. From total order to database replication. In *Proceedings of the* 22^{nd}*IEEE International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002. IEEE Computer Society Press.

[AW94] H. Attiya and J. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[BBG89] C. Beeri, P. Bernstein, and N. Goodman. A model for concurrency in nested transactions systems. *Journal of the ACM*, 36(2):230–269, 1989.

[BBG$^+$95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In M. Carey and D. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 1–10. ACM Press, 1995.

[BGMS89] D. Barbara, H. Garcia-Molina, and A. Spauster. Increasing availability under mutual exclusion constraints with dynamic vote reassignment. *ACM Transactions on Computer Systems*, 7(4):394–426, 1989.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[Bir85] K. Birman. Replication and fault-tolerance in the isis system. *SIGOPS Operating Systems Review*, 19(5):79–86, 1985.

[BMST93] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems*, chapter 8, pages 199–216. Addison-Wesley, second edition, 1993.

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

[BvR94] K. Birman and R. van Renesse. *Reliable distributed computing with the Isis toolkit*. IEEE Computer Society Press, 1994.

[CAA92] S. Cheung, M. Ammar, and M. Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):582–592, 1992.

[CMZ03] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Raidb: Redundant array of inexpensive databases. Technical Report 4921, INRIA - Rhône-Alpes, September 2003.

[CMZ04] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *USENIX Annual Technical Conference, FREENIX Track*, pages 9–18, 2004.

[CNO99] J. Cowie, D. Nicol, and A. Ogielski. Modeling the global Internet. *Computing in Science and Engineering*, 1(1), January/February 1999.

[Cow99] J. Cowie. *Scalable Simulation Framework API Reference Manual*, March 1999.

[Cri91] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

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

[DM96] D. Dolev and D. Malki. The transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996.

[DSS98] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of 17^{th} IEEE International Symposium on Reliable Distributed Systems*, pages 43–50, West lafayette, IN, USA, October 1998. IEEE Computer Society Press.

[DSU04] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[EL75] P. Erdös and L. Lovasz. Problems and results on 3-chromatic hypergraphs and some related questions. In A. Hajnal et al., editors, *Infinite and Finite Sets*, volume 11 of *Colloq. Math. Soc. Janos Bolyai*, pages 609–627. North-Holland, 1975.

[EMS95] P. Ezhilchelvan, R. Macêdo, and S. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15^{th}IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society Press, May/June 1995.

[ES83] D. Eager and K. Sevcik. Achieving robustness in distributed database systems. *ACM Transactions on Database Systems,* 8(3):354–381, 1983.

[FLO$^+$05] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems,* 30(2):492–528, 2005.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[GG02] G.Weikum and G.Vossen. *Transactional Information systems.Theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., 2002.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, June 1996.

[Gif79] D. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the 7^{th} symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM Press.

[GMB85] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, 1985.

[GR93] J. Gray and A. Reuter. *Transaction Processing: concepts and techniques*. Data Management Systems. Morgan Kaufmann Publishers Inc., San Francisco, California, 1993.

[GS97a] R. Guerraoui and A. Schiper. Total order multicast to multiple groups. In *Proceedings of the 17^{th}IEEE International Conference on Distributed Computing Systems (ICDCS)*, page 578. IEEE Computer Society Press, May 1997.

[GS97b] Rachid Guerraoui and André; Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[GSC$^+$83] N. Goodman, D. Skeen, A. Chan, U. Dayal, S. Fox, and D. Ries. A recovery algorithm for a distributed database system. In *Proceedings of the 2^{nd} ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 8–15. ACM Press, March 1983.

[Gue95] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *Proceedings of the 9^{th} International Workshop on Distributed Algorithms (WDAG-9)*, volume 972 of *Lecture Notes in Computer Science*, pages 87–100, Le Mont-St-Michel, France, September 1995. Springer.

[HAA99] J. Holliday, D. Agrawall, and A. El Abbadi. The performance of database replication with group multicast. In *Proceedings of the* 29^{th}*IEEE International Symposium on Fault-Tolerant Computing*, pages 158–165. IEEE Computer Society Press, June 1999.

[HAA00] J. Holliday, D. Agrawall, and A. El Abbadi. Database replication using epidemic communication. In *Europar 2000*, 2000.

[HAA02] J. Holliday, D. Agrawal, and A. El Abbadi. Partial database replication using epidemic communication. In *Proceedings of the* 22^{nd}*IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 485–493. IEEE Computer Society Press, July 2002.

[Her87] M. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems*, 12(2):170–194, 1987.

[HLvR99] J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for Ensemble layers. In R. Cleaveland, editor, *TACAS '99: Proceedings of the* 5^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 119–133. Springer, March 1999.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed systems (2nd Ed.)*, chapter 5, pages 97–147. ACM Press/Addison-Wesley Publishing Co., 1993.

[IOZ04] IOzone filesystem benchmark. http://www.iozone.org, 2004.

[JM90] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, 15(2):230–280, 1990.

[JPPMAK03] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems*, 28(3):257–294, 2003.

[JPPMKA02] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings of the* 14^{th}*IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 477–484. IEEE Computer Society Press, July 2002.

[Jún04] A. Júnior. Distributed transaction processing in the escada protocol. Master's thesis, Universidade do Minho, Departamento de Informática - Escola de Engenharia - Universidade do Minho, May 2004.

[KA00a] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *VLDB '00: Proceedings of the* 26^{th}*International Conference on Very Large DataBases (VLDB)*, pages 134–143. Morgan Kaufmann Publishers Inc., September 2000.

[KA00b] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, 2000.

[KPAS99] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proceedings of the* 19^{th}*IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 424–431. IEEE Computer Society Press, June 1999.

[KRS93] A. Kumar, M. Rabinovich, and R. Sinha. A performance study of general grid structures for replicated data. In *Proceedings of the* 13^{th}*IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 178–185. IEEE Computer Society Press, May 1993.

[KT91] M. Kaashoek and A. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the* 11^{th}*IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society Press, May 1991.

[Kum91] A. Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Transactions on Computers*, 40(9):996–1004, 1991.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[MAMSA94] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended virtual synchrony. In *Proceedings of the* 14^{th}*IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society Press, June 1994.

[MMSA$^+$96] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: a fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, 1996.

[MPR01] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the* 21^{st}*IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 707–710. IEEE Computer Society Press, April 2001.

[MRR06] J. Mocito, A. Respício, and L. Rodrigues. On statistically estimated optimistic delivery in wide-area total order protocols. In 12^{th}*Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE Computer Society Press (To appear), 2006.

[MSMA90] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, 1990.

[Nei96] G. Neiger. A new look at membership services (extended abstract). In *PODC '96: Proceedings of the* 15^{th} *annual ACM symposium on Principles of distributed computing*, pages 331–340. ACM Press, 1996.

[NW98] M. Naor and A. Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.

[OPS01] R. Oliveira, J. Pereira, and A. Schiper. Primary-backup replication: From a time-free protocol to a time-based implementation. In *Proceedings of the* 20^{th}*IEEE International Symposium on Reliable Distributed Systems*. IEEE Computer Society Press, October 2001.

[Pâr86] J. Pâris. Voting with witnesses: A constistency scheme for replicated files. In *Proceedings of the* 6^{th}*IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 606–612. IEEE Computer Society Press, May 1986.

[Pâr89] J. Pâris. Voting with bystanders. In *Proceedings of the* 9^{th}*IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 394–405. IEEE Computer Society Press, June 1989.

[Pax97] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, Univ. of CA, Berkeley, April 1997.

[PBS89] L. Peterson, N. Buchholz, and R. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3), August 1989.

[PCD91] D. Powell, M. Chéréque, and D. Drackley. Fault-tolerance in delta-4*. *ACM Operating Systems Review, SIGOPS Conference*, 25(2):122–125, April 1991.

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

[Pet04] M. Pettersson. Linux performance counters. http://user.it.uu.se/ mikpe/linux/perfctr/, 2004.

[PGP05] C. Le Pape, S. Gançarski, and P. Replica refresh strategies in a database cluster. In *BDA*, 2005.

[PGS98] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'98)*, Southampton, England, September 1998.

[PGS03] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distribributed and Parallel Databases*, 14(1):71–98, 2003.

[PL91] J. Pâris and D. Long. Voting with regenerable volatile witnesses. In *Proceedings of the* 7^{th}*IEEE International Conference on Data Engineering*, pages 112–119. IEEE Computer Society Press, April 1991.

[PMJPKA00] M. Patiño-Martínez, R. Jiménez-Paris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proceedings of the* 14^{th} *Internationnal conference on Distributed Computing (DISC 2000)*, volume 1914 of *Lecture Notes in Computer Science*, pages 315–329. Springer, October 2000.

[PO05] J. Pereira and R. Oliveira. Rewriting "the turtle and the hare": Sleeping to get there faster. In *Proceedings of the First Workshop on Hot Topics in System Dependability*, June 2005.

[PS99] F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the* 13^{th} *International Symposium on Distributed Computing (DISC'99, formerly WDAG)*, volume 1693 of *Lecture Notes in Computer Science*. Springer, September 1999.

[PS03] F. Pedone and A. Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291(1):79–101, 2003.

[PSQ] PostgreSQL. http://www.postgresql.org.

[PW97] D. Peleg and A. Wool. Crumbling walls: a class of practical and efficient quorum systems. *Distributed Computing*, 10(2):87–97, 1997.

[Ric93] A. Ricciardi. *The group membership problem in asynchronous systems*. PhD thesis, Cornell University, Ithaca, NY, USA, 1993.

[RMC06] L. Rodrigues, J. Mocito, and N. Carvalho. From spontaneous total order to uniform total order: different degrees of optimistic delivery. In *Proceedings of the* 21^{st} *ACM Symposium on Applied Computing (SAC'06)*. ACM, April 2006.

[SAA98] I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of the* 18^{th}*IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 148–155. IEEE Computer Society Press, May 1998.

[Sch93] F. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed systems (2nd Ed.)*, chapter 7. ACM Press/Addison-Wesley Publishing Co., 1993.

[SPOM01] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *IEEE International Symposium on Network Computing and Applications*. IEEE Computer Society Press, October 2001.

[SR96] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39:84–87, 1996.

[SSF04] Scalable simulation framework. http://www.ssfnet.org, 2004.

[SWWW00] R. Schenkel, G. Weikum, N. Weißenberg, and X. Wu. Federated transaction management with snapshot isolation. In *Selected papers from the Eight International Workshop on Foundations of Models and Languages for Data and Objects, Transactions and Database Dynamics*, pages 1–25, London, UK, 2000. Springer-Verlag.

[TGGL82] I. Traiger, J. Gray, C. Galtieri, and B. Lindsay. Transactions and consistency in distributed database systems. *ACM Transactions on Database Systems*, 7(3):323–342, 1982.

[Tho79] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[TPC01] TPC Benchmark$^{\text{TM}}$ C standard specification revision 5.0, February 2001.

[TPK95] O. Theel and H. Pagnia-Koch. General design of grid-based data replication schemes using graphs and a few rules. In *Proceedings of the* 15^{th}*IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 395–403. IEEE Computer Society Press, May/June 1995.

[vRT88] R. van Renesse and A. Tanenbaum. Voting with ghosts. In *Proceedings of the* 8^{th}*IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 456–462. IEEE Computer Society Press, June 1988.

[WB92] C. Wu and G. Belford. The triangular lattice protocol: A highly fault tolerant and highly efficient protocol for replicated data. In *Proceedings of the* 11^{th}*IEEE International Symposium on Reliable Distributed Systems*, pages 66–73. IEEE Computer Society Press, October 1992.

[WK05] S. Wu and B. Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *IEEE International Conference on Data Engineering*, pages 422–433. IEEE Computer Society Press, April 2005.

[WPS^{+}00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In

Proceedings of the 20^{th}*IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 264–274. IEEE Computer Society Press, April 2000.