Replication Using Group Communication Over a Partitioned Network

Thesis submitted for the degree "Doctor of Philosophy"

Yair Amir

Submitted to the Senate of the Hebrew University of Jerusalem (1995).

This work was carried out under the supervision of

Professor Danny Dolev

Acknowledgments

I am deeply grateful to Danny Dolev, my advisor and mentor. I thank Danny for believing in my research, for spending so many hours on it, and for giving it the theoretical touch. His warm support and patient guidance helped me through. I hope I managed to adopt some of his professional attitude and integrity.

I thank Daila Malki for her help during the early stages of the Transis project. Thanks to Idit Keidar for helping me sharpen some of the issues of the replication server. I enjoyed my collaboration with Ofir Amir on developing the coloring model of the replication server. Many thanks to Roman Vitenberg for his valuable insights regarding the extended virtual synchrony model and the replication algorithm. I benefited a lot from many discussions with Ahmad Khalaila regarding distributed systems and other issues. My thanks go to David Breitgand, Gregory Chokler, Yair Gofen, Nabil Huleihel and Rimon Orni, for their contribution to the Transis project and to my research.

I am grateful to Michael Melliar-Smith and Louise Moser from the Department of Electrical and Computer Engineering, University of California, Santa Barbara. During two summers, several mutual visits and extensive electronic correspondence, Louise and Michael were involved in almost every aspect of my research, and unofficially served as my co-advisors. The work with Deb Agarwal and Paul Ciarfella on the Totem protocol contributed a lot to my understanding of high-speed group communication.

Ken Birman and Robbert van-Renesse from the Computer Science Department at Cornell University, were always willing to contribute their valuable advice to my research. Spending last summer with them was an educating experience for me. For that I thank them both. Special thanks to Ken for convincing me to pursue an academic position.

Thanks to Eldad Zamler for first introducing me to what became my research problem, ten years ago. I thank Yaacov Ben-Yaacov and Gidi Kuperstein for six years of collaboration in building a working system and delivering it to the customer. They are all special friends.

I would like to thank my parents Shulamit and Reuven, for their love, encouragement and constant support. I thank my brother Yaron, my brother Ofir, Amira and Lee, for always being there for me.

Last, but not least, I am grateful to my wife and my partner Michal, for her unending support. My success is the product of her wisdom, confidence, and love.

Contents

Abstract

In systems based on the client-server model, a single server may serve many clients and the heavy load on the server may cause the response time to be adversely affected. In such circumstances, replicating data or servers may improve performance. Replication may also improve the availability of information when processors crash or the network partitions.

Existing replication methods are often needlessly expensive. They sometimes use point-to-point communication when multicast communication is available; they typically pay the full price of end-to-end acknowledgments for all of the participants for every update; they may claim locks, and therefore, may be vulnerable to faults that can unnecessarily block the system for long periods of time.

This thesis presents a new architecture and algorithms for replication over a partitioned network. The architecture is structured into two layers: a replication server and a group communication layer. Each of the replication servers maintains a private copy of the database. Actions (queries and updates) requested by the application are globally ordered by the replication servers in a symmetric way. Ordered actions are applied to the database and result in a state change and in a reply to the application.

We provide a group communication package, named Transis, to serve as the group communication layer. Transis utilizes the available non-reliable hardware multicast for efficient dissemination of messages to a group of processes. The replication servers use Transis to multicast actions and to learn about changes in the membership of the currently connected servers, in a consistent manner. Transis locally orders messages sent within the currently connected servers. The replication servers use this order to construct a long-term global total order of actions.

Since the system is subject to partitioning, we must ensure that two detached components do not reach contradictory decisions regarding the global order. Therefore, the replication servers use dynamic linear voting to select, at most, one primary component that continues to order actions.

The architecture is non-blocking: actions can be generated by the application anytime. While in a primary component, queries are immediately replied in a consistent manner. While in a non-primary component, the user can choose to wait for a consistent reply (that will arrive as soon as the network is repaired) or to get an immediate, though not necessarily consistent reply.

High performance of the architecture is achieved because:

- End-to-end acknowledgments are not needed on a regular basis. They are used only after membership change events such as processor crashes and recoveries, and network partitions and merges.
- Synchronous disk writes are almost eliminated, without compromising consistency.
- Hardware multicast is used where possible.

Chapter 1

1. Introduction

In systems based on the client-server model, a single server may serve many clients and the heavy load on the server may cause the response time to be adversely affected. In such circumstances, replicating data or servers may improve performance. Replication may also improve the availability of information when processors crash or the network partitions.

Existing replication methods are often needlessly expensive. They sometimes use point-to-point communication when multicast communication is available. They typically pay the full price of end-to-end acknowledgment for all of the participants for every update, or even of several rounds of end-to-end acknowledgments. They may claim locks, and therefore, may be vulnerable to faults that can unnecessarily block the system for long periods of time.

This thesis ends a ten year professional journey. It started with my involvement in the design and implementation of a large and geographically distributed control system. The requirements of that system demanded a non-blocking solution with maximal availability. Each of the control stations had to be autonomous, to work despite network partitions, and to survive power failures. To meet the requirements, we constructed a data replication scheme to function over an unreliable communication media in a dynamic environment. We managed to limit the update semantics to commutative updates. Hence, the replica control problem was reduced to implementing a guaranteed delivery of actions to all of the replicas. This was done by constructing point-to-point stable queues. The concept was proven adequate and is still operational today, maintaining consistent replication of several tens of databases. However, the use of point-to-point communication and the extensive use of synchronous disk writes, as well as the limitation imposed on the update semantics, left me with a feeling that a better replication concept can be found. My Ph.D. research was motivated by this belief.

Together with Danny Dolev, Dalia Malki and Shlomo Kramer, we initiated the Transis system, targeted at building tools for highly available distributed systems. We gave Transis its name to acknowledge the innovation of both the Trans protocol [MMA90] and the ISIS system [BvR94]. Transis was aimed at providing group communication services using non-reliable hardware multicast available in most local area networks, tolerating network partitions and merges as well as processor crashes and recoveries.

On top of Transis, we designed a replication server that eliminates the need for synchronous disk writes per update without compromising consistency. Avoiding disk writes on the critical path and utilizing hardware multicast renders our replication architecture highly efficient and more scalable than previous solutions.

1.1 Problem Description

The problem tackled in this thesis is how to construct an efficient and robust long-term replication architecture, within a fixed set of servers. Each server maintains a private copy of the database. The initial state of the database is identical at all of the servers. Typically, each server runs on a different processor.

The replication architecture is required to handle network partitioning. We explicitly assume that the network may partition to several components. Some or all of the partitioned components, may subsequently re-merge. The architecture is also required to handle server crashes and recoveries. It is assumed that the underlying communication supports some form of non-reliable multicast service (this service can be mimicked by unreliable point-to-point transmission). The architecture is required to overcome message omissions.

We assume no message corruption. We rely on error detection and error correction protocols to eliminate corrupted messages. Corrupted messages have the effect of omitted messages.

We do not handle malicious faults. We assume that all the servers are running their protocols faithfully.

1.2 Solution Highlights

We present a new architecture and algorithms for active replication over a partitioned network. Active replication is a symmetric approach where each of the replicas is guaranteed to invoke the same set of actions at the same order. This approach requires the next state of the database to be determined by the current state and the next action, and it guarantees that all of the replicas reach the same database state. Other factors, such as the passage of time, should not have any bearing on the next database state.

The architecture, presented in Figure 1.1, is structured into two layers: a replication server and a group communication layer. Each of the replication servers maintains a private copy of the database. Actions (queries and updates) requested by the application are globally ordered by the replication servers in a symmetric way. Ordered actions are applied to the database and result in a state change and in a reply to the application.

The replication servers use the group communication layer to efficiently disseminate actions, and to learn about changes in the membership of the currently connected servers in a consistent manner. The group communication layer locally orders messages disseminated within the currently connected group.

When a new component is formed by merging two or more components, the servers exchange information about actions and about the actions' order in the system. Actions missed by at least one of the servers, are multicast, and the connected servers reach a

common state. This way, actions are propagated as soon as possible. We call this method *propagation by eventual path*.

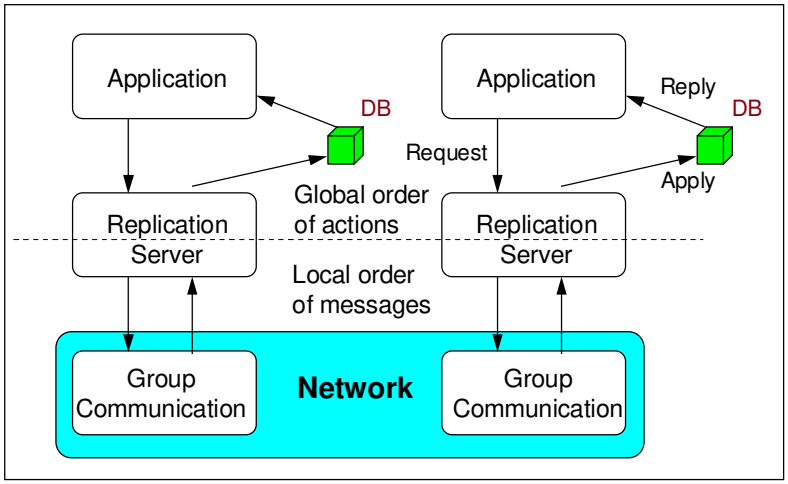

Figure 1.1: The Architecture.

Since the system may partition, we must ensure that two different components do not reach contradictory decisions regarding the global order of actions. Hence, we need to identify at most one component, the *primary* component, that may continue ordering actions. We employ dynamic linear voting [JM90] which is generally accepted as the best technique when certain restrictions hold.

We define a new semantics, *extended virtual synchrony*, for the group communication service. The significance of extended virtual synchrony is that, during network partitioning and re-merging and during process crash and recovery, it maintains a consistent relationship between the delivery of messages and the delivery of configuration change notifications across all processes in the system.

Prior group communication protocols have focused on totally ordering messages at the group communication level. That service, although useful for some applications, is not enough to guarantee complete consistency at the application level without additional end-to-end acknowledgments, as has been noted by Cheriton and Skeen [CS93]. Extended virtual synchrony specifies the *safe delivery* service which provides additional level of knowledge within the group communication protocol.

The strict semantics of extended virtual synchrony and its safe delivery service is exploited by the replication servers to eliminate the need for end-to-end acknowledgment on a per-action basis **without** compromising consistency. End-to-end acknowledgment is only required when the membership of connected servers is changed. e.g. in case of network partitions, merges, server crashes and recoveries.

This leads to high performance of the architecture. In the general case, when the membership of connected servers is stable, the throughput and latency of actions is

determined by the performance of the group communication and not so much by other factors such as the number of replicas and the performance of synchronous disk writes.

The architecture is non-blocking: actions can be generated by the application anytime. While in a primary component, queries are immediately replied in a consistent manner. While in a non-primary component, the user can choose to wait for a consistent reply (that will arrive as soon as the network is repaired) or to get an immediate, though not necessarily consistent reply. Two different, well-defined, semantics are available for immediate replies in a non-primary component.

The key contributions of this Ph.D. research are:

- Defining an efficient architecture for replication.
- Constructing a highly efficient reliable multicast protocol that tolerates partitions, and implementing it in a general Unix environment. The symmetric protocol provides reliable message ordering and membership services. The protocol's exceptional performance is achieved by utilizing a non-reliable multicast service where possible.
- Defining the extended virtual synchrony semantics for group communication services. Extended virtual synchrony, among other things, strictly defines message delivery semantics in the presence of network partitions and re-merges, as well as process crashes and recoveries.
- Constructing the propagation by eventual path technique for efficient information dissemination in a dynamic network. This method utilizes group communication to propagate knowledge as soon as possible between servers. The strengths of the propagation by eventual path method are most evident when the membership of connected servers is dynamically changing.
- Eliminating the need for end-to-end acknowledgments and for synchronous disk writes on a per-action basis. Instead, end-to-end acknowledgments and synchronous disk writes are needed once, just after a change in the membership of the connected servers.
- Tailoring and optimizing replication services for different kinds of applications.

1.3 Thesis Organization

The rest of the thesis is organized as follows:

- The next subsection presents previous research in group communication protocols, group communication semantics, and replication protocols.
- Chapter 2 presents the theoretical model and defines the correctness criteria of the solution.

- Chapter 3 presents the overall replication architecture.
- Chapter 4 defines the extended virtual synchrony semantics.
- Chapter 5 presents Transis, our group communication layer, which provides extended virtual synchrony. We describe the logical ring protocol, one of the two reliable multicast protocols operational in Transis. Throughput and latency measurements of Transis, over a network of Pentium machines running Unix, are provided.
- Chapter 6 details our replication server. The replication protocol demonstrates how extended virtual synchrony is exploited to provide efficient long-term replication service.
- Chapter 7 customizes services for different kinds of applications.
- Chapter 8 concludes this thesis.

A reader, interested in an overview of this thesis beyond the introduction, may read Chapter 3, Chapter 5 Section 1 and Section 3, and Chapter 6 Section 1.

A reader interested in the practical aspects of this thesis and in implementation details, may want to focus on Chapter 3, Chapter 5 Section 2 and Section 3, Chapter 6 Section 2, and Chapter 7.

Additional information including a copy of this thesis, a slide show, relevant published papers and more, can be obtained from:

http://www.cs.jhu.edu/yairamir **or**
http://www.cs.huji.ac.il/~dolev
or by writing to yairamir@cs.jhu.edu

1.4 Related Work

Much work has been done in the area of group communication and in the area of replication. We relate our work to three research areas: group communication protocols, group communication semantics, and replication protocols.

1.4.1 Group Communication Protocols

The ISIS toolkit [BJ87, BCJM+90, BvR94] is one of the first general purpose group communication systems. ISIS provides a group communication session service, where processes can join process groups, multicast messages to groups, and receive messages sent to groups. Two multicast primitives are provided: The CBCAST service guarantees causally ordered message delivery (see [Lam78]) across overlapping groups. CBCAST is implemented using vector timestamps that are piggybacked on each message. The ABCAST service extends the causal order to a total order using a central group coordinator that emits ordering decisions. ISIS also provides membership notifications

when the group membership is changed. Group membership changes due to processes voluntarily joining or leaving the group, or due to process failures. Network partitions and re-merges, as well as process recoveries, are not supported. The novelty of ISIS is in guaranteeing a formal and rigorous service semantics named *virtual synchrony*. ISIS protocols are implemented using point-to-point communication. Although much better protocols exist today, and despite the lack of support for network partitions, ISIS is the most mature general purpose system available today. The ISIS system is commercially available from ISIS Distributed Systems LTD.

The V system [CZ85] provides group communication services at the operating system level. It was the first to utilize hardware multicast to implement process group communication. However, only non-reliable, best-effort, unordered delivery service is provided. Similar services for wide area networks are provided by the IP-multicast [Dee89] protocol.

The Chang and Maxemchuk reliable broadcast and ordering protocol [CM84] uses a token-passing strategy, where the processor holding the token acknowledges messages. All the participating processors can broadcast messages at any time. The protocol also provides membership and token recovery algorithms. Typically, between two and three messages are required to order a message in an optimally loaded system. The protocol does not provide a mechanism for flow control.

The TPM protocol [RM89] uses a token on a logical ring of processors for broadcasting and retransmission of messages. The token is circulated along a known token list in order to serialize message transmission. The token contains the next sequence number to be stamped on new messages. TPM starts by circulating the token to multicast a set of messages. Then, the token is used to retransmit messages belonging to the set, that are missed by some of the processors. When no message is missed by any of the processors, the whole set is delivered to the application and a new set of messages can be introduced. TPM also provides a dynamic membership and token regeneration algorithm. If the network partitions, the component with the majority of the members (if such exists) is allowed to continue.

The Delta-4 [Pow91] system provides tools for building distributed, fault-tolerant real-time systems. As part of Delta-4, a reliable multicast protocol, *x*AMp [RV92] and a membership protocol [RVR93] are implemented. The protocols utilize the non-reliable multicast or broadcast primitive of local area networks. The Delta-4 protocols assume fail-stop behavior and as such, do not support network partitions and re-merges. The membership protocol provides low-level processor membership so that a higher level process group membership can be built on top of it in a simple way. Our experience in Transis indicates that this two-levels architecture is better than solving the membership problem at the process level. Delta-4 is more real-time oriented than Transis, and it uses a special hardware for message ordering and failure detection. This seems to be a strong limitation on the project's usability.

The Amoeba distributed operating system uses the Flip high performance reliable multicast protocol [KvRvST93] to support high level services such as fault-tolerant directory service [KTV93]. In Amoeba, members of the group send point-to-point messages to a distinct member called the sequencer. The sequencer stamps each message with a sequence number and broadcasts it to the group. A Member that detects a gap in the message sequences, sends a point-to-point retransmission request to the sequencer. The Amoeba system is resilient to any pre-defined number of failed processors, but its performance degrades as the number of allowed failures is increased.

The Trans and Total protocols [MMA90, MMA93, MM93] provide reliable ordered broadcast delivery in an asynchronous environment. The Trans protocol uses positive and negative acknowledgments piggybacked onto broadcast messages and exploits the transitivity of positive acknowledgments to reduce the number of acknowledgments required. The Total protocol, layered on top of the Trans protocol, converts the partial order into a total order. The Trans and Total protocols maintain causality and ensure that operational processors continue to order messages even though other processors have failed, provided that a resiliency constraint is met. A membership protocol [MMA94] is implemented on top of Total. If a processor suspects another processor, it sends a fault message for the suspected processor. When that message is ordered, the membership is changed to exclude this processor. The limitation of that architecture is that if Total cannot order the membership messages (e.g. because the resiliency constraint is not met), the system is blocked.

The Psync protocol [PBS89] builds a context graph that represents the causal partial order on messages. This order can be extended into a total order by determining complete waves of causally concurrent messages and by ordering the messages of a wave using some deterministic order. Based on the causal order provided by Psync, a membership algorithm is constructed [MPS91]. Using this algorithm, processors reach eventual agreement on membership changes. The algorithm handles processor faults and allows a processor to join a pre-existing group asymmetrically. Network partitions and re-merges are not supported.

The Newtop protocol [MES93, Mac94] replaces the context graph of Psync by the notion of causal blocks. Each causal block defines a set of messages. All the messages within a block are causally independent. The blocks are totally ordered. The messages in a block are delivered together, in some deterministic order. In this way, Newtop provides totally ordered delivery similar to the wave technique of Psync and the all-ack mechanism of Lansis [ADKM92a], but with much less bookkeeping. Newtop causal delivery is less efficient than Psync or Trans because the causal information represented in causal blocks is not accurate and more pessimistic then needed (though more compact). Moreover, using causal blocks eliminates the ability to use faster algorithms (e.g. TOTO [DKM93]) that use the full context graph to reach fast decision on total order. Newtop implements a membership service that handles processor crashes and network partitions. However, process recoveries and network re-merges are not addressed. The most interesting point of Newtop is its service semantics presented in the next section.

The Horus project [vRBFHK95] implements group communication services, providing unreliable or reliable FIFO, causal, or total multicast services. Horus is extensively layered

and highly configurable, allowing applications to only pay for the overhead of services they use. The layers include the COM layer which provides basic non-reliable multicast, the NAK layer which provides reliable FIFO multicast, the MBRSHIP layer that provides membership maintenance, the STABLE layer which provides message stability, the FC layer which provides flow control, the CAUSAL and TOTAL layers, the LWG layer which maintains process groups, the EVS layer which maintains extended virtual synchrony (see below), and many more. Advanced memory management techniques are used in order to avoid the full cost of layering.

The Transis project, described in Section 5.1, provides group communication services in a partitionable network. Three multicast primitives are provided according to the *extended virtual synchrony* semantics: Causal multicast, Agreed multicast for total order delivery, and Safe multicast that provides even stronger guarantees. Two different reliable multicast protocols are implemented in Transis. Lansis [ADKM92a], the earlier protocol, uses a direct acyclic graph (DAG) representing the causal relation on messages to provide reliable multicast. The DAG is derived from negative and positive acknowledgments piggybacked on messages. The causal order mechanism in Lansis is derived from the Trans protocol with several important modifications that adapt it for practical use. Two total order algorithms extended the causal order to a total, agreed order. The first is the all-ack algorithm which is similar to the algorithm used in Psync, and the second is the TOTO early delivery algorithm [DKM93]. Both computes the total order based on the DAG structure without exchange of additional messages. While TOTO is more efficient than the all-ack protocol, it cannot maintain extended virtual synchrony.

The membership algorithm of Transis [ADKM92b] is a symmetric protocol that was the first to handle network partitions and re-merges. Although operational in asynchronous environment, the algorithm ensures termination in a bounded time. The basic idea of this membership algorithm was adopted by Totem and Horus. Excellent reading about Transis and its membership algorithm is found in [Mal94].

The second reliable multicast protocol in Transis is the Ring protocol, detailed in Section 5.2. The Ring protocol was developed while the author was visiting the Totem project.

The Totem system [Aga94] provides reliable multicast and membership services across a collection of local-area networks. The Totem system is composed of a hierarchy of two protocols. The bottom layer is the Ring protocol [AMMAC93, AMMAC95] which provides reliable multicast and processor membership services within a broadcast domain. The upper layer is the Multiple-Rings protocol [Aga94] that provides reliable delivery and ordering across the entire network. Gateways are responsible to forward messages and configuration changes between broadcast domains. Each gateway interconnects two broadcast domains, and participates in the Ring protocol for each of them. Each domain may contain several gateways connecting it to several other domains. Extended virtual synchrony was first implemented in the Totem system [AMMAC93].

1.4.2 Group Communication Semantics

It is highly important for a group communication service to maintain a well-defined service semantics. The application builder can rely on that semantics when designing correct applications using this group communication service. The semantics must specify both the assumptions taken and the guarantees provided.

The ISIS system defines and maintains the *virtual synchrony* semantics [BvR94, BJ87, SS93]. Virtual synchrony ensures that all the processes belonging to a process group perceive configuration changes as occurring at the same logical time. Moreover, all processes belonging to a configuration deliver the same set of message for that configuration. A message is guaranteed to be delivered at the same configuration in which it was multicast at all the processes that deliver it. The delivery of a CBCAST message maintains causality. The delivery of an ABCAST message, in addition, occurs at the same logical time at all the processes.

Virtual synchrony assumes message omission faults and fail-stop process faults. i.e. a process that fails can never (or is not allowed to) recover. When network partitioning occurs, virtual synchrony ensures that processes in at most one connected component of the network, the primary component, are able to make progress; processes in other components become blocked.

Unfortunately, before a process fails or before it detects that it had partitioned from the primary component, ISIS may deliver messages to it in an order inconsistent with the order determined at the primary component (if a database is maintained by the detached process, these messages may result in an inconsistent database state). Therefore, if a process recovers after a crash, or can merge again with the primary component, it must come back with a different process identifier and it is considered as a new process. If this process maintains stable storage (e.g. database), this storage has to be erased.

Unable to cope with network partitions and re-merges, and with process recoveries, virtual synchrony has a limited practical value. Nevertheless, the virtual synchrony model emphasized the importance of a rigorous semantics for group communication services. To overcome these drawbacks, we extended the definition of virtual synchrony. This extension, *extended virtual synchrony* [MAMA94] is detailed in Chapter 4.

Valuable work done at the Newtop project [Mac94], separately from the work done in Transis and Totem, defines another group communication semantics which extends virtual synchrony to support partitions. Newtop semantics specifies several properties regarding the delivery of messages and configuration changes. It generalizes the primary component model of virtual synchrony to support several partitioned components without the need to block non-primary components (the application is, of course, free to block operation in non-primary components if it prefers). Newtop semantics is weaker than the extended virtual synchrony semantics. In particular, since Newtop does not support network re-merges, weaker requirements are specified for totally ordered delivery. This weakness allows the total order determined at a process to vary, and to contain holes, when compared to the total order determined at another process that just partitioned. Moreover,

Newtop semantics does not specify the *safe delivery* property of extended virtual synchrony, whose importance is made clear at Chapter 6 of this thesis.

A recent work by Cristian and Schmuck on group membership in an asynchronous environment [CS95] defines the *timed synchronous system model*. In contrast to the theoretical asynchronous model that has no notion of time, the timed synchronous model assumes that processors have local clocks that allow them to measure the passage of time. Local clocks may drift with some (small) bounded rate. Each processor also contains a stable storage. Processor crashes introduce partial-amnesia behavior where the state of stable storage is the same as before the crash, while the state of the volatile storage is reinitialized. The model allows for message omission or performance (delay) faults, processor crashes and recoveries, and network partitions and re-merges. The unique aspect of [CS95], lays in bounding the local time up to which certain guarantees of the group membership service will hold at each of the processors. While the membership algorithms developed in Transis and Totem **do** maintain the requirements presented in [CS95], they are not **required** to do so by the extended virtual synchrony model (which leaves local time out of the model).

Combining ideas from the timed synchronous model to extended virtual synchrony might lead to a model which guarantees stronger liveness properties (that are provided anyway by the implementations of Transis and Totem). This, in turn, might lead to the ability to prove stronger liveness properties (with bounded local time) for protocols that currently use extended virtual synchrony to reason about their behavior. e.g. it might be possible to prove a better liveness property for the replication protocol described in Chapter 6, than the required liveness property stated in Chapter 2.

1.4.3 Replication Protocols

Much work has been done in the area of replication. Traditionally, a replicated database is considered correct if it behaves as if there is only one copy of it, as far as the user can tell. This property is called *one-copy equivalence*. In a one-copy database, the system should ensure *serializability*. i.e. interleaved execution of user transactions is equivalent to some serial execution of these transactions. Thus, a replicated database is considered correct if it is *one-copy serializable* ([BHG87]). i.e. it ensures serializability and one-copy equivalence.

Two-phase-commit protocols [EGLT76] are the main tool for providing serializability in a distributed database system when transactions may span several sites. The same protocols can be used to maintain one-copy serializability in a replicated database. In a typical protocol of this kind [Gra78], one of the servers, the transaction coordinator, sends a request to prepare to commit to all of the participating servers. Each server replies either by a “ready to commit” or by an “abort”. If any of the servers votes to abort, all of them abort. The transaction coordinator collects all the responses and informs the servers of the decision. Between the two phases, each server keeps the local database locked waiting for the final word from the transaction coordinator. If a server fails before its vote reaches the

transaction coordinator, it is usually assumed to vote "abort". If the transaction coordinator fails, all the servers remain blocked indefinitely, unable to resolve the transaction. Even though blocking preserves consistency, it is highly undesirable because the locks cannot be relinquished, rendering the data inaccessible by other requests at operational servers. Clearly, a protocol of this kind imposes a substantial additional communication cost on each transaction.

Three-phase-commit protocols [Ske82] try to overcome some of the availability problems of two-phase-commit protocols, paying the price of an additional communication round, and therefore, of additional latency. In case of server crashes or network partitions, a three-phase-commit protocol allows a majority or a quorum to resolve the transaction. If failures cascade, however, a majority can be connected and still remain blocked as is shown in [KD95]. A recent work by [KD95] presents an improved version of three-phase-commit that **always** allows a connected majority to proceed, regardless of past failures.

In the available copy protocols [BHG87], update operations are applied at all of the available servers, while a query accesses any server. Correct execution of these protocols require that the network never partition. Otherwise they block.

Voting protocols are based on quorums. The basic quorum scheme uses majority voting [Tho79] or weighted majority voting [Gif79]. Using voting protocols, each site is assigned a number of votes. The database can be updated in a partition only if that partition contains more than half of the votes.

The Accessible Copies algorithms [ESC85, ET86] maintain an approximate view of the connected servers, called *a virtual partition*. A data item can be read/written within a virtual partition only if this virtual partition (which is an approximation of the current connected component) contains a majority of its read/write votes. If this is the case, the data item is considered accessible and read/write operations can be done by collecting sub-quorums in the current component. The maintenance of virtual partitions greatly complicates the algorithm. When the view changes, the servers need to execute a protocol to agree on the new view, as well as to recover the most up-to-date item state. Moreover, although view decisions are made only when the "membership" of connected servers changes, each update requires the full end-to-end acknowledgment from the sub-quorum.

Dynamic linear voting [JM87, JM90] is a more advanced approach that defines the quorum in an adaptive way. When a network partition (or re-merge) occurs, if a majority of the last installed quorum is connected, a new quorum is established and updates can be performed within this partition. Dynamic linear voting generally outperforms the static schemes as shown by [PL88].

Epsilon serializability [PL91] applies an extension to the serializability correctness criterion. Epsilon serializability introduces a tradeoff between consistency and availability. It allows inconsistent data to be seen, but requires that data will eventually converge to a consistent (one-copy serializability) state. The user can control the degree of inconsistency. In the limit, strict one-copy serializability can be enforced. Several replica control protocols are suggested in [PL91]. One of these protocols limits the transactional model to commutative operations (COMMU) and another limits it to read-independent

timestamped updates (RITU). In contrast, the ordered updates (ORDUP) protocol does not limit the transactional model. ORDUP executes transactions asynchronously, but in the same order at all of the replicas. Update transactions are disseminated and are applied to the database when they are totally ordered. The replication protocol presented in Chapter 6 of this thesis complies with the ORDUP model. Optimizations for COMMU and RITU updates models are presented in Chapter 7 of this thesis.

Lazy replication [LLSG90, LLSG92] is a replication method that overcomes network partitions and re-merges. It relaxes the constraints on operation ordering by exploiting the semantics of the service's operations. The client application can specify exactly what causal relations should be enforced between operations. Using this approach, unrelated operations do not incur any latency delay due to communication. By using a gossip method to propagate operations, lazy replication ensures reliable eventual delivery of all the operations to all of the replica. However, the loose control on operation transmissions between replicas is a serious drawback of lazy replication. An operation might be transmitted from one replica to another many times, even when it is already known at the other replica.

The timestamped anti-entropy replication technique [Gol90] provides eventual weak consistency. This method also ensures the eventual delivery of each action to each of the replication servers using an epidemic technique: Pairs of servers periodically contact each other to exchange actions that one of them has and the other misses. This exchange is called anti-entropy session. When the network partitions and subsequently re-merges, servers from different components exchange actions generated at the disconnected component using anti-entropy sessions. A total order on the actions can be placed using a similar method to [AAD93]. The anti-entropy technique used to propagate actions is far more efficient compared to the gossip technique of [LLSG90].

In prior research [AAD93], we described an architecture that uses the Transis group communication layer to achieve consistent replication. The architecture handles network partitions and re-merges, as well as server crashes and recoveries. It constructs a highly efficient epidemic technique, using the configuration change notification provided by Transis to keep track of the membership of the currently connected servers. Upon a reconfiguration change, the currently connected servers efficiently exchange state information. Each action known to one of the servers and missed by at least one server, is sent exactly once. The replication servers does not need to worry about message omissions because the group communication layer (Transis) guarantees reliable multicast. This technique is more efficient than the anti-entropy technique because instead of using two-way exchange of knowledge and actions, multi-way exchange is used. Moreover, the exchange takes place exactly when it is needed (i.e. after a membership change) rather than periodically. The serious inefficiency of [AAD93] is the method of global total ordering, which uses Lamport clock and requires an eventual path from every server to order an action.

A valuable work by Keidar [Kei94] uses the architecture of [AAD93] but replaces its global total ordering method. The novel ordering algorithm in [Kei94] **always** allows a connected majority of the servers to make progress, regardless of past failures. As in [AAD93], it always allows servers to initiate actions (even when they are not part of a

connected majority). Thus, actions can eventually become totally ordered even if their initiator is never a member of a majority component.

Both [Kei94] and [AAD93] use the flow control and multicast properties of group communication, but both still need an end-to-end acknowledgments between servers on a per-action basis to allow global ordering of a message. This diminishes the performance advantages gained by using group communication.

The replication server, described in [ADMM94] and detailed in Chapter 6 of this thesis, eliminates the need for an end-to-end acknowledgment at servers level **without** compromising consistency. End-to-end acknowledgment is still needed just after the membership of the connected server is changed. Thus, the performance gain is substantial, and is determined by the performance provided by the group communication. The price to pay (compared to [Kei94]) is that there exist rare scenarios in which multiple servers in the primary component crash or become disconnected within a window of time so short that the membership algorithm could not be completed anywhere. In these scenarios, if none of the servers is certain about which actions were ordered within that primary component (e.g. due to a global crash), then the recovery of, and communication with, every server of the last primary component is required before the next primary component can be formed.

Chapter 2

2. The Model

2.1 The Service Model

A *Database* is a collection of organized, related data that can be accessed and manipulated. An *Action* defines a transition from the current state of the database to the next state; the next state is completely determined by the current state and the action. Each Action contains an optional *query* part and an optional *update* part. The update part of an action defines a modification to be made to the database, and the query part returns a value.

A *replication service* maintains a replicated database in a distributed system. The replication service is provided by a known finite set of processes, called the *servers group*. The individual processes within the servers group are called *replication servers* or simply *servers*, each of which has a unique identifier. Each server within the servers group maintains a private copy of the database on stable storage. The initial state of the database is identical at all of the servers. Typically, each server runs on a different processor.

Processes to which the service is provided are called *clients*. The number of clients in the system is unlimited.

We introduce the following notation:

- S is the servers group.
- $a_{s,i}$ is the *i*th action performed by server *s*.
- $D_{s,i}$ is the state of the database at server *s* after actions 1..*i* have been performed by server *s*.
- *stable_system*(*s, r*) is a predicate that denotes the existence of a set of servers containing *s* and *r*, and a time, from which on, that set does not face any communication or server failure. Note that this predicate is only defined to reason about the liveness of certain protocols. It does not imply any limitation on our practical protocol.

2.2 The Failure Model

The system is subject to message omission, server crashes and network partitions. We assume no message corruption and no malicious faults.

A server or a processor may crash and may subsequently recover after an arbitrary amount of time. A server recovers with its stable storage intact, is aware of its recovery, and retains its old identifier.

The network may partition into a finite number of *components*. The servers in a component can receive messages generated by other servers in the same component, but servers in two different components are unable to communicate with each other. Two or more components may subsequently merge to form a larger component.

A message which is multicast within a component may get lost by some or even all of the processors.

2.3 Replication Requirements

According to the service model, the initial state of the database is identical at all of the servers.

$$\forall s, r \in S \qquad D_{s,0} = D_{r,0}.$$

Also, the next state of the database is completely determined by the current state and the performed action.

$$\forall s \in S \qquad D_{s,i} = function\left(D_{s,i-1}, a_{s,i}\right).$$

The correctness criteria for the solution are defined as follows:

- **Safety.** If server *s* performs the *i*th action and server *r* performs the *i*th action, then these actions are identical.

  $$\exists a_{s,i}, a_{r,i} \Rightarrow a_{s,i} = a_{r,i}.$$

  Note that if the servers perform the same set of actions in the same order then they reach an identical state. For databases that comply with our service model (where the next database state is completely determined by the current state and the performed action), our safety criterion translates to one-copy serializability (see [BHG87]). One-copy serializability requires that concurrent execution of actions on a replicated database be equivalent to some serial execution of these actions on a non-replicated database.

- **Liveness.** If server *s* performs an action and there exists a set of servers containing *s* and *r*, and a time, from which on, that set does not face any communication or processes failures, then server *r* eventually performs the action.

  $\Diamond(\exists a_{s,i} \wedge \Box\, stable\_system(s,r)) \Rightarrow \Diamond \exists a_{r,i}$.

  Our liveness criterion only admits protocols that propagate actions between any two servers, while it excludes protocols that rely on a central server, or on some specific servers, to propagate actions.

Chapter 3

3. The Architecture

Two main approaches for replication are known in the literature: the first is the *primary-backup* approach, and the second is *active replication*.

In the primary-backup approach, one of the replication servers, the *primary*, is the only server allowed to respond to application requests (actions). The other servers, the *backups*, update their copy of the database after the primary informs them of the action. If the primary crashes, one of the backups takes over and becomes the new primary. Some primary-backup architectures allow backups to respond to queries in order to increase system performance.

Active replication, in contrast, is a symmetric approach where each of the replication servers is guaranteed to invoke the same set of actions in the same order. This approach requires the next database state to be determined by the current state and the next action. Other factors, such as the passage of time, have no bearing on the next state. Some active replication architectures replicate only the updates, while queries are locally replied.

This work takes the approach of active replication. As can be seen in Figure 3.1, our replication architecture is a symmetric architecture which is structured into two layers: a replication server layer and a group communication layer. Typically, each replication server is a process that runs on a different processor that hosts a copy of the database. The group communication layer is another process running on the same processor and communicating with the replication server via inter process communication mechanisms. Alternatively, it can be implemented as a library which is linked within the replication server process.

Each of the replication servers maintains a private copy of the database. The client application *requests* an action from one of the replication servers. The client-server interaction is done via some communication mechanism such as RPC, IPC, or even via the group communication layer. The replication servers agree on the order of actions to be performed on the replicated database. As soon as a replication server knows the final order of an action, it *applies* this action to the database. If the action contains a query part, a *reply* is returned to the client application from the database copy maintained by the original server that got the request. The replication servers use the group communication layer to disseminate the actions among the servers group and to help reach an agreement about the final global order of the set of actions.

In a typical operation, when an application requests an action from a replication server, this server *generates* a message containing the action. The message is then passed to the local group communication layer which *sends* the message over the communication medium. Each of the **currently connected** group communication layers finally *receives* the message and then *delivers* the message in the same order to their replication servers. We say that these servers are currently connected**.**

If the system partitions into several components, the replication servers identify at most one component as the *primary* component. The replication servers in a primary component determine the final global total order of actions according to the order provided by the group communication layer. As soon as the final order of an action is determined, this action is applied to the database. In the primary component, new actions can be ordered, and be applied to the database, immediately upon delivery by the group communication layer. In non-primary components, actions must be delayed until communication is restored and the servers learn of the order determined by the primary component.

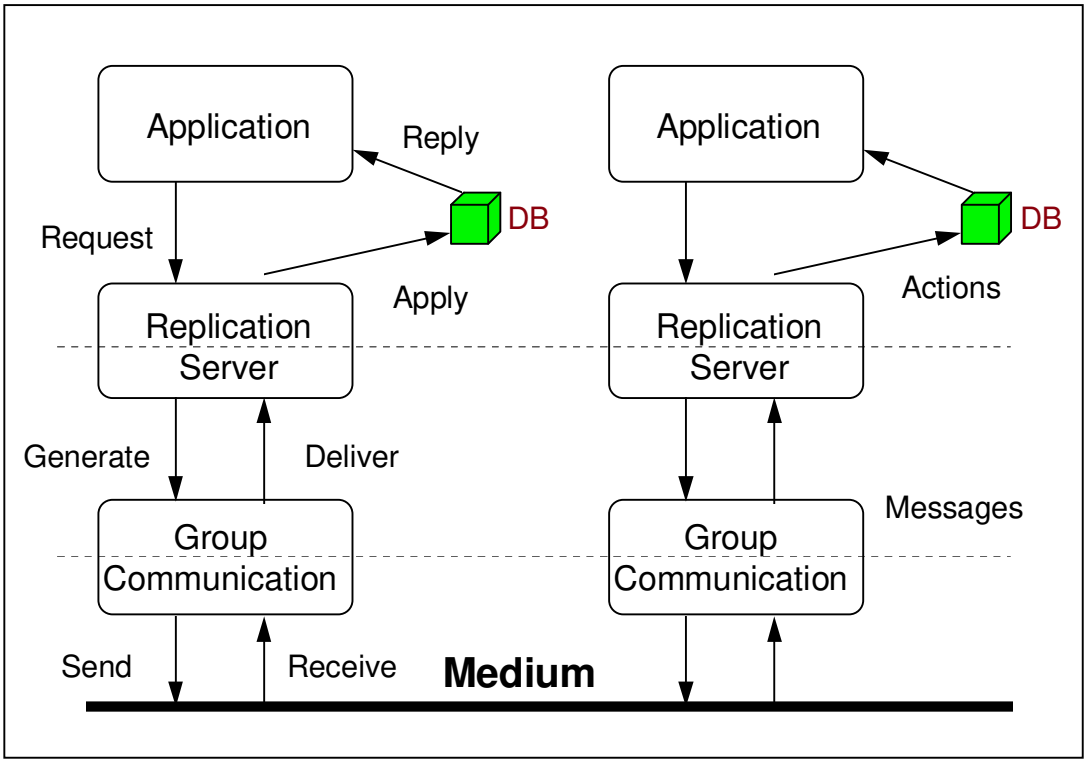

Figure 3.1: Detailed Architecture

The group communication layer provides reliable multicast and membership services according to the extended virtual synchrony model specified in Chapter 4. This layer overcomes message omission faults and notifies the replication server of changes in the membership of the currently connected servers. This notification corresponds to server crashes and recoveries and to network partitions and re-merges. The Transis system, which is an implementation of such group communication layer is described in Chapter 5.

On notification of a membership change by the group communication layer, the replication servers exchange messages containing actions sent before the membership change. This exchange of information ensures that every action known to a member of the currently connected servers becomes known to all of them. Moreover, knowledge of final order of actions is also shared among the currently connected servers. As a consequence, after this exchange is completed, the state of the database at each of the connected servers is identical. A detailed description of the replication server is given in Chapter 6.

Our experience with developing distributed applications for different environments is that the main difficulties in developing such applications arise from the asynchronous communication exchange and from failure handling, while maintaining consistency. These difficulties is almost completely handled by Transis. We have found, for example, that developing a reliable mail service without our group communication layer required seven times the code length. Moreover, we have found that the most problematic portion, facing the asynchronous nature of processor crashes and network partitions, is almost eliminated. It is true that this ratio depends on the application, but the same principle applies to many distributed applications. Once we have given the application developer a clean interface to communicate and handle failures, the code becomes simpler, faster to develop and probably better performing.

The same principle is applied to the structure we have used within our replication architecture. We have separated the group communication layer from the replication server and by that we have simplified the replication server. Beyond that, once we had established that separation, the issue of when one needs to use end-to-end acknowledgments was crystallized. It became clear that there is no need to apply end-to-end acknowledgments on a per-action basis. As long as no membership change takes place, nothing prevents us from eventually reaching consistency. The only careful handling of message exchange and order verification are needed once a membership change takes place. Our protocol reflects this observation.

Chapter 4

4. Extended Virtual Synchrony

This chapter specifies a semantics for a group communication transport layer. A group communication layer that maintains extended virtual synchrony guarantees to comply with this semantics subject to the failure model described in Chapter 2. This chapter is based on joint work with Louise Moser, Michael Melliar-Smith and Deb Agarwal [MAMA94] while the author visited the Totem project.

Extended virtual synchrony extends the virtual synchrony model of the Isis system [BvR94]. Virtual synchrony in Isis is designed to support failures that respect the fail-stop failure model. In addition, extended virtual synchrony supports crash and recovery failures and network partitions and re-merges.

The significance of extended virtual synchrony is that, during network partitioning and re-merging and during process crash and recovery, it maintains a consistent relationship between the delivery of messages and the delivery of configuration change notifications across all processes in the system. Moreover, extended virtual synchrony maintains well-defined self-delivery and failure atomicity properties.

Each processor that may have processes participating in the group communication runs one group communication daemon or layer (GC) such as Transis. Each GC executes a reliable multicast and membership algorithm such as the one described in Chapter 5. The physical communication is handled by the GC. The membership algorithm determines the processes that are members of the current component. This membership, together with a unique identifier, is called a *configuration*. A configuration which is installed by a process, represents this process' view of the connectivity in the system. The membership algorithm ensures that all processes in a configuration agree on the membership of that configuration. Each process is informed of changes in the configuration by the delivery of configuration change messages.

As was discussed is the previous chapter, we distinguish between *receipt* of a message by the GC over the communication medium, which may be out of order, and *delivery* of a message by the GC to the process, which may be delayed until prior messages in the order have been delivered. Messages can be delivered in agreed order and in safe order. *Agreed delivery* guarantees a total order of message delivery within each component and allows a message to be delivered as soon as all of its predecessors in the total order have been delivered. *Safe delivery* requires in addition, that if a message is delivered by the GC to any of the processes in a configuration, this message has been received and will be delivered to each of the processes in the configuration unless it crashes.

To achieve safe delivery in the presence of network partitioning and re-merging, and of process crash and recovery, extended virtual synchrony presents two configuration types. In a *regular configuration* new messages are sent and delivered. In a *transitional configuration* no new messages are sent but the remaining messages from the prior regular configuration are delivered. Those messages did not satisfy the safe delivery requirements in the regular configuration, and thus, could not be delivered there. A transitional configuration consists of members of the next regular configuration coming directly from the same regular configuration.

A regular configuration may be immediately followed by several transitional configurations (one for each component of the partitioned network) and may be immediately preceded by several transitional configurations (when several components merge together). A transitional configuration, in contrast, is immediately followed by a single regular configuration and is immediately preceded by a single regular configuration (because it consists **only** of members of the next regular configuration coming directly from the same regular configuration).

For a process p that is a member of a regular configuration c, we define $trans_p(c)$ to be the transitional configuration that follows c at p, if such a configuration exists. For a process p that is a member of a transitional configuration c, $trans_p(c) = c$. For a process p that is a member of a transitional configuration c, we define $reg(c)$ to be the regular configuration that immediately precedes c. For a process p that is a member of a regular configuration c, $reg(c) = c$. We define $com_p(c)$ to be either one of the configurations $reg(c)$ or $trans_p(c)$. Note that if both p and q are members of c, $trans_p(c)$ is not necessarily equal to $trans_q(c)$ and, thus, $com_p(c)$ is not necessarily equal to $com_q(c)$.

Extended virtual synchrony is defined in terms of four types of events:

- $deliver\_conf_p(c)$: the GC delivers to process p a configuration change message initiating configuration c where p is a member of c.
- $send_p(m,c)$: the GC sends message m generated by p while p is a member of configuration c.
- $deliver_p(m,c)$: the GC delivers message m to p while p is a member of configuration c.
- $crash_p(c)$: process p crashes or the processor at which p resides crashes while p is a member of configuration c.

The $crash_p(c)$ event is the actual failure of process p in configuration c and is distinct from a $deliver\_conf_q(c')$ event that removes p from configuration c at process q. After a $crash_p(c)$ event, process p may remain failed forever, or may recover with a $deliver\_conf_p(c'')$ where the configuration c'' is $\{p\}$.

The precedes relation, $\rightarrow$, defines a global partial order on all events in the system. The *ord* function, from events to natural numbers defines a logical total order on those events. The *ord* function is not one-to-one, because some events in different processes are required to occur at the same logical time. The semantics of extended virtual synchrony below define the $\rightarrow$ relation and the *ord* function.

4.1 Extended Virtual Synchrony Semantics

The semantics of extended virtual synchrony consists of Specifications 1-7 below. In the figures, vertical lines correspond to processes, an open circle represents an event that is assumed to exist, a star represents an event that is asserted to exist, a light edge without an arrow represents a precedes relation that holds because of some other specification, a medium edge with an arrow represents a precedes relation that is assumed to hold between two events, a heavy edge with an arrow represents a precedes relation that is asserted to hold between two events, and a cross through an event (relation) indicates that the event (relation) does not occur. In all the figures, time increases downwards.

4.1.1 Basic Delivery

Specification 1.1 requires that the $\rightarrow$ relation is an irreflexive, anti-symmetric and transitive partial order relation. Specification 1.2 requires that the events within a single process are totally ordered by the $\rightarrow$ relation. Specification 1.3 requires that the sending of a message precedes its delivery, and that the delivery occurs in the configuration in which the message was sent or in an immediately following transitional configuration. Specification 1.4 asserts that a given message is not sent more than once and is not delivered in two different configurations to the same process.

1.1 For any event e, $e \not\rightarrow e$.
If there exist events e and e', such that $e \rightarrow e'$, it is not the case that $e' \rightarrow e$.
If there exist events e, e' and e'' such that $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

1.2 If there exists an event e that is $deliver\_conf_p(c)$ or $send_p(m,c)$ or $deliver_p(m,c)$ or $crash_p(c)$, and an event e' that is $deliver\_conf_p(c')$ or $send_p(m',c')$ or $deliver_p(m',c')$ or $crash_p(c')$, then $e \rightarrow e'$ or $e' \rightarrow e$.

1.3 If there exists $deliver_p(m,c)$, then there exists $send_q(m, reg(c))$ such that $send_q(m, reg(c)) \rightarrow deliver_p(m,c)$.

1.4 If there exists $send_p(m,c)$, then $c = reg(c)$ and there is neither $send_p(m,c')$ where $c \neq c'$, nor $send_q(m,c'')$ where $p \neq q$.

Moreover, if there exists $deliver_p(m,c)$, then there does not exist $deliver_p(m,c')$ where $c \neq c'$.

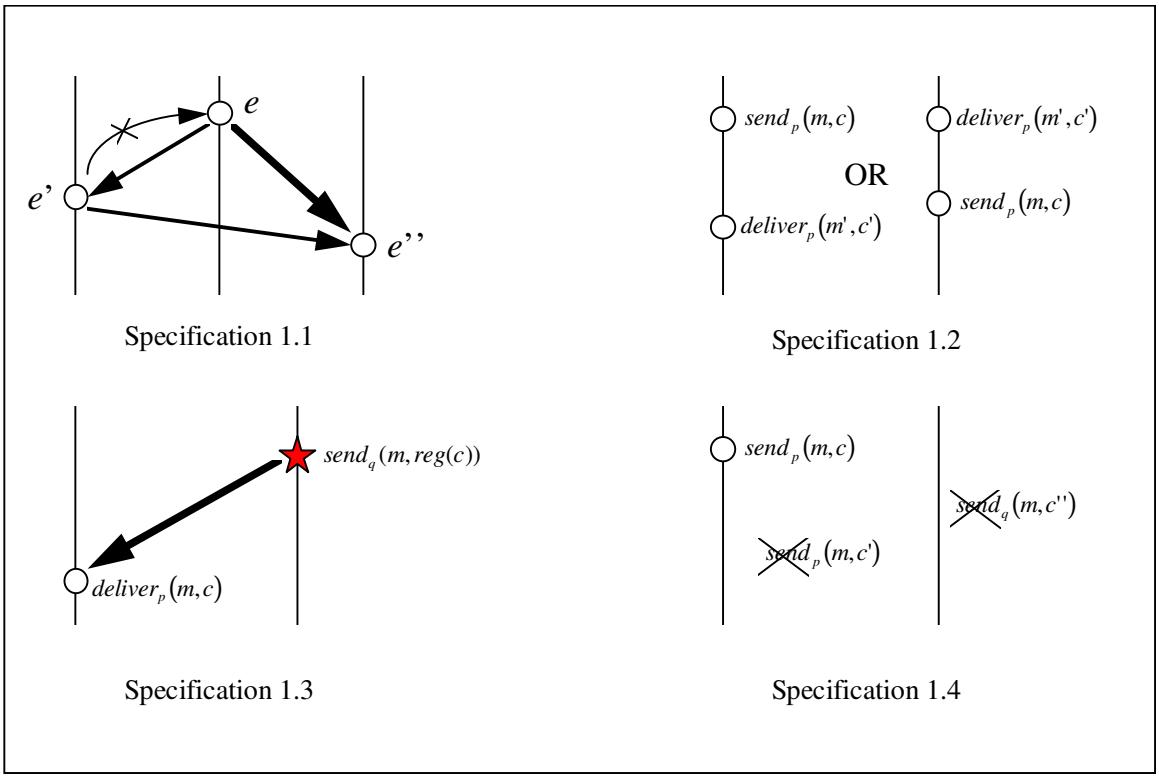

Figure 4.1: Basic Delivery Specifications

4.1.2 Delivery of Configuration Changes

Specification 2.1 requires that if a process crashes or partitions, then the GC detects that and delivers a new configuration change message to other processes belonging to the old configuration. Specification 2.2 states that at any moment a process is a member of a unique configuration whose events are delimited by the configuration change event(s) for that configuration. Specifications 2.3 and 2.4 assert that an event that precedes (follows) delivery of a configuration change to one process must also precede (follow) delivery of that configuration change to other processes.

2.1 If there exists $deliver\_conf_p(c)$ and there does not exist $crash_p(c)$ and there does not exist $deliver\_conf_p(c')$ such that $deliver\_conf_p(c) \rightarrow deliver\_conf_p(c')$, and if q is a member of c, then there exists $deliver\_conf_q(c)$, and there does not exist $crash_q(c)$,and there does not exists $deliver\_conf_q(c'')$ such that $deliver\_conf_q(c) \rightarrow deliver\_conf_q(c'')$.

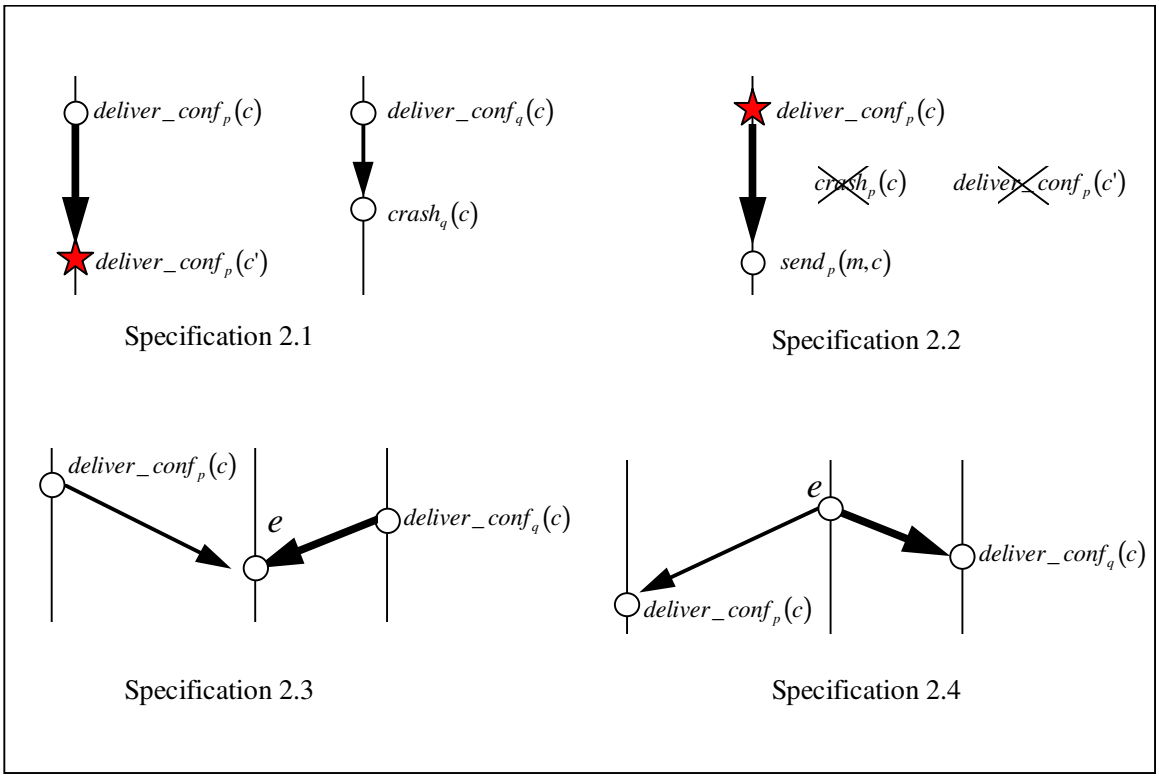

Figure 4.2: Configuration Change Specifications

2.2 If there exists an event e that is either $send_p(m,c)$, $deliver_p(m,c)$, or $crash_p(c)$, then there exists $deliver\_conf_p(c)$ such that $deliver\_conf_p(c) \rightarrow e$, and there does not exist an event e' such that e' is $crash_p(c)$ or $deliver\_conf_p(c')$ and $deliver\_conf_p(c) \rightarrow e' \rightarrow e$.

2.3 If there exist $deliver\_conf_p(c)$, $deliver\_conf_q(c)$ and e such that $deliver\_conf_p(c) \rightarrow e$, then $deliver\_conf_q(c) \rightarrow e$.

2.4 If there exist $deliver\_conf_p(c)$, $deliver\_conf_q(c)$ and e such that $e \rightarrow deliver\_conf_p(c)$, then $e \rightarrow deliver\_conf_q(c)$.

4.1.3 Self Delivery

Specification 3 requires that each message that is generated by a process is delivered to this process, provided that it does not crash. Moreover, the message is delivered in the same configuration it was sent, or in the transitional configuration which follows.

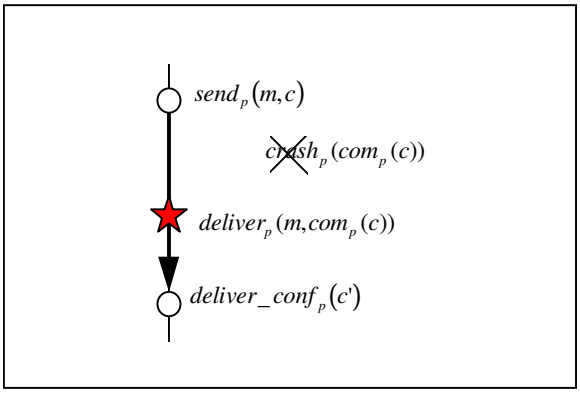

Figure 4.3: Self Delivery Specification

3. If there exist $send_p(m,c)$ and $deliver\_conf_p(c')$ where $c' \neq trans_p(c)$, such that $send_p(m,c) \rightarrow deliver\_conf_p(c')$, and there does not exist $crash_p(com_p(c))$, **then** there exists $deliver_p(m,com_p(c))$

4.1.4 Failure Atomicity

Specification 4 requires that if any two processes proceed together from one configuration to the next, the GC delivers the same set of messages to both processes in that configuration.

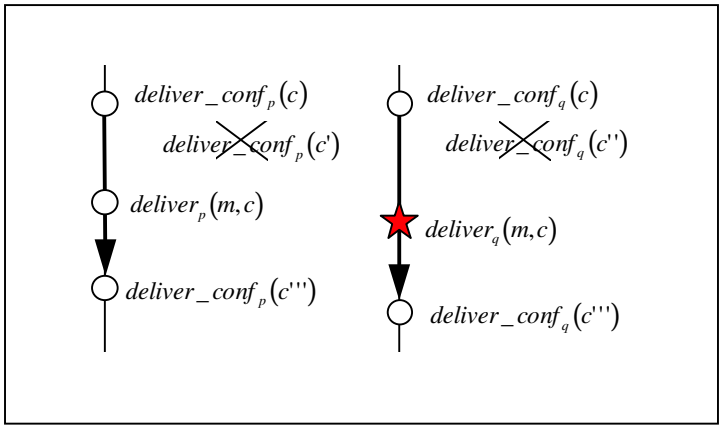

Figure 4.4: Failure Atomicity Specification

4. If there exist $deliver\_conf_p(c)$, $deliver\_conf_p(c''')$, $deliver\_conf_q(c)$, $deliver\_conf_q(c''')$ and $deliver_p(m,c)$, such that $deliver\_conf_p(c) \rightarrow deliver\_conf_p(c''')$, and there does not exist $deliver\_conf_p(c')$ such that $deliver\_conf_p(c) \rightarrow deliver\_conf_p(c') \rightarrow deliver\_conf_p(c''')$ and there does not exist $deliver\_conf_q(c'')$ such that $deliver\_conf_q(c) \rightarrow deliver\_conf_q(c'') \rightarrow deliver\_conf_q(c''')$ **then** there exists $deliver_q(m,c)$.

4.1.5 Causal Delivery

We model causality so that it is local to a single configuration and is terminated by a configuration change message. Simpler formulations of causality are not appropriate [Lam78, BvR94] when a network may partition and re-merge or when a process may crash and recover. The causal relationship between messages is expressed in Specification 5 as a precedes relation between the sending of two messages in the same configuration. This precedes relation is contained in the transitive closure of the precedes relations established by Specifications 1.1-1.3.

Specification 5 requires that if one message is sent before another in the same configuration and if the GC delivers the second of those messages, then it also delivers the first.

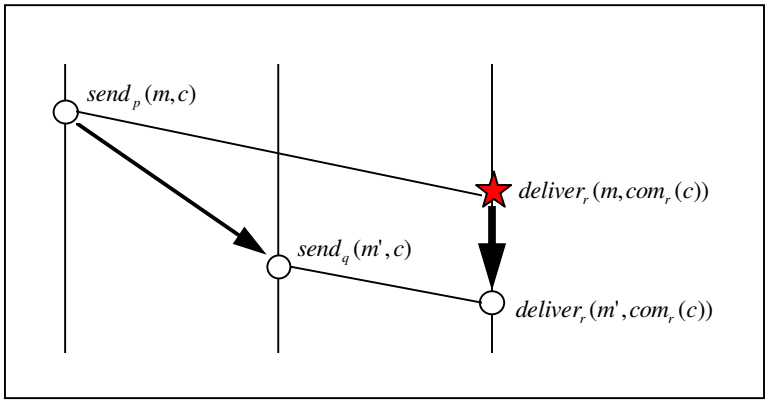

Figure 4.5: Causal Delivery Specification

5. If there exist $send_p(m,c)$, $send_q(m',c)$ and $deliver_r(m',com_r(c))$ such that $send_p(m,c) \rightarrow send_q(m',c)$, then there exists $deliver_r(m,com_r(c))$ such that $deliver_r(m,com_r(c)) \rightarrow deliver_r(m',com_r(c))$.

4.1.6 Agreed Delivery

The following specifications contain the definition of the *ord* function. Specification 6.1 requires the total order to be consistent with the partial order. Specification 6.2 asserts that the GC delivers configuration change messages for the same configuration, at the same logical time to each of the processes. Messages are also delivered at the same logical time to each of the processes, regardless of the configuration in which they are delivered. Specification 6.3 requires that the GC delivers messages in order to all processes except that, in the transitional configuration there is no obligation to deliver messages generated by processes that are not members of that transitional configuration.

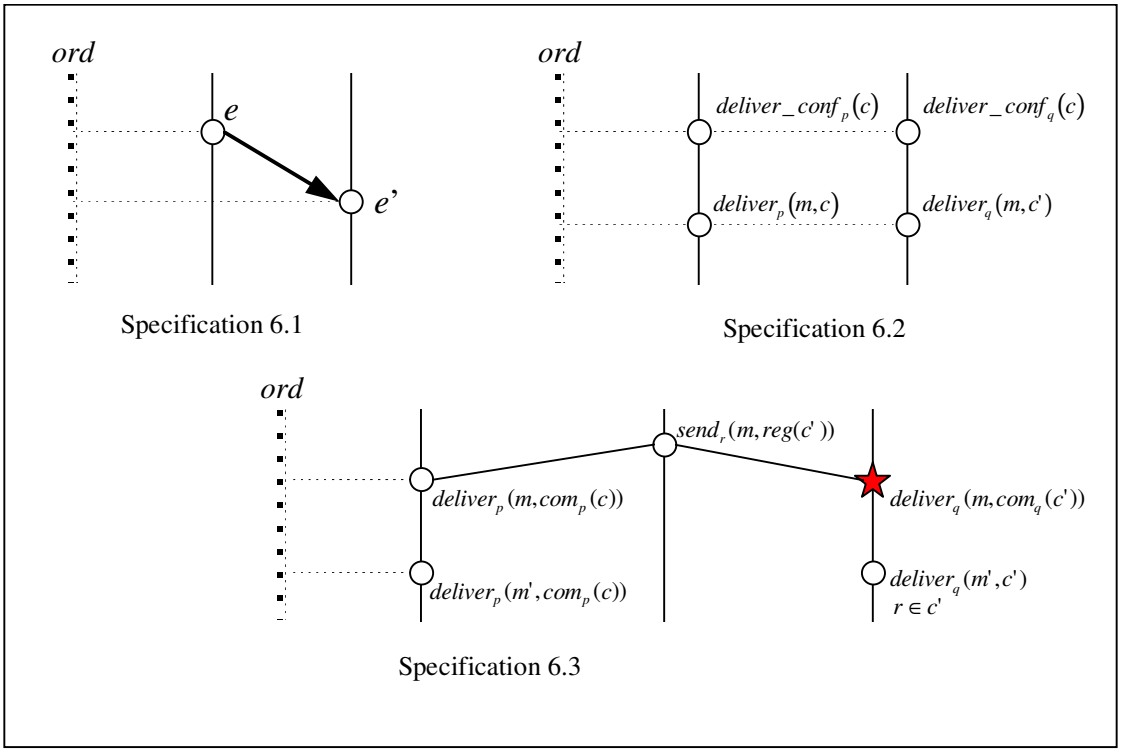

Figure 4.6: Totally Ordered Delivery Specifications

6.1 If there exist events e and e' such that $e \rightarrow e'$, then $ord(e) < ord(e')$.

6.2 If there exist events e and e' that are either $deliver\_conf_p(c)$ and $deliver\_conf_q(c)$ or $deliver_p(m,c)$ and $deliver_q(m,c')$, then $ord(e) = ord(e')$.

6.3 If there exist $deliver_p(m,com_p(c))$, $deliver_p(m',com_p(c))$, $deliver_q(m',c')$ and $send_r(m,reg(c'))$ such that $ord(deliver_p(m,com_p(c))) < ord(deliver_p(m',com_p(c)))$ and r is a member of c', then there exists $deliver_q(m,com_q(c'))$.

Note that the relationship between c and c' in Specification 6 can only be one of the following: either they are the same regular or transitional configuration or they are different transitional configurations for the same regular configuration, or one is a regular configuration and the other is a transitional configuration that follows it.

4.1.7 Safe Delivery

Specification 7.1 requires that, if the GC delivers a safe message to a process which is in a configuration, then the GC delivers the message to each of the processes in that configuration unless the process crash. i.e. even if the network partitions at that point, the message is still delivered. Specification 7.2 asserts that, if the GC delivers a safe message to any of the processes in a regular configuration, then the GC delivered the configuration change message for that configuration to all the members of that configuration.

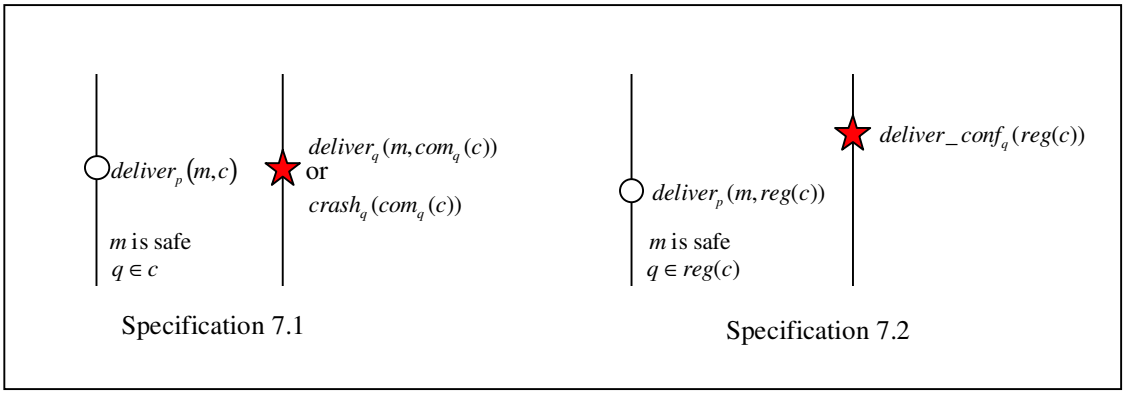

Figure 4.7: Safe Delivery Specifications

7.1 If there exists $deliver_p(m,c)$ for a safe message m, then for every process q in c there exists either $deliver_q(m,com_q(c))$ or $crash_q(com_q(c))$.

7.2 If there exists $deliver_p(m,reg(c))$ for a safe message m, then for every process q in $reg(c)$ there exists $deliver\_conf_q(reg(c))$.

4.2 An Example of Configuration Changes and Message Delivery

Consider the example shown in Figure 4.8. Here, a regular configuration containing *p*, *q* and *r* partitions and *p* becomes isolated while *q* and *r* merge into a new regular configuration with *s* and *t*. While still in { *p, q, r* }, five safe messages were sent at the following order: *m*1 was sent by *p*, *m*2 was sent by *q*, *m*3 was sent by *p*, *m*4 was sent by *r* and *m*5 was sent by *p*. *p, q* and *r* can deduce that *m*1 was received by all of them.

At *p*, all five messages were received. *p* can deduce that *q* and *r* have received *m*1 and *m*2. Therefore, *m*1 and *m*2 meet the safe delivery requirements and are delivered at *p* in the regular configuration { *p, q, r* }. However, *p* cannot tell whether *m*3, *m*4 and *m*5 were received by all members of { *p, q, r* }. Therefore, a transitional configuration { *p* } is delivered at *p* followed by *m*3, *m*4, *m*5 and by the next regular configuration { *p* }.

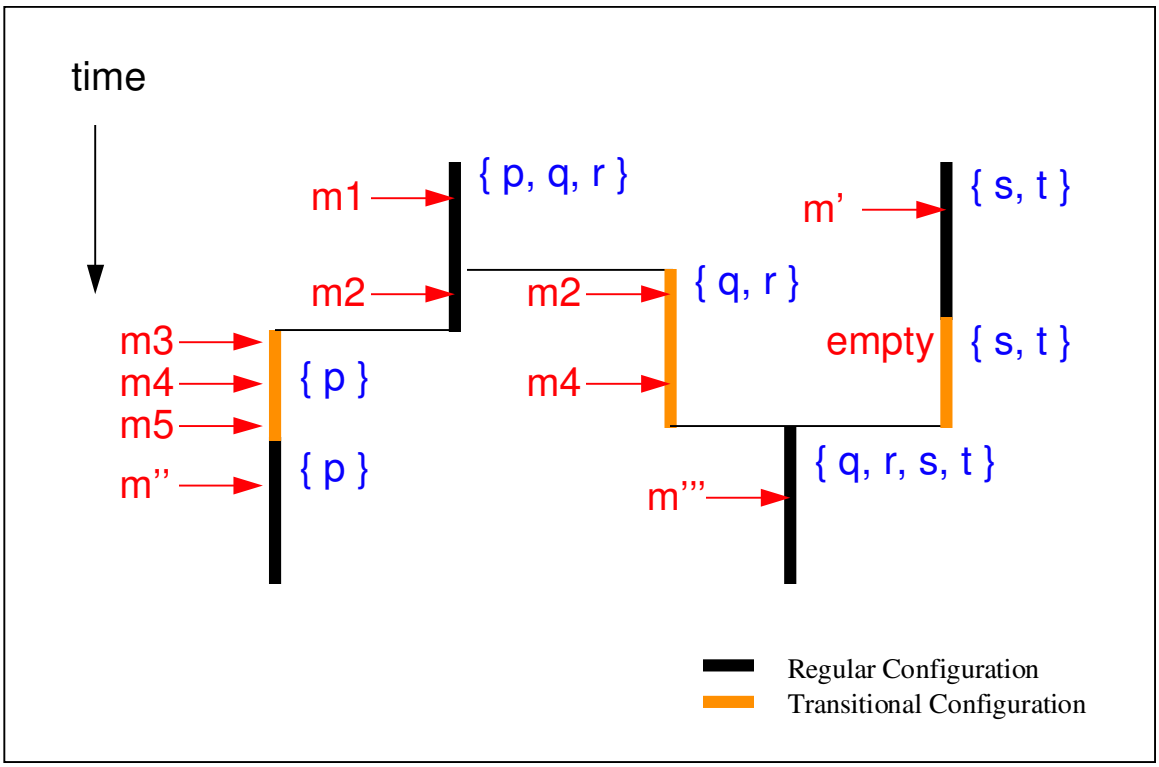

Figure 4.8: An Example of Configuration Changes and Message Delivery

At *q* and *r*, only four messages were received: *m*1, *m*2, *m*4 and *m*5. Since *q* and *r* know that *p* is required to deliver *m*1, *m*1 meets the safe delivery requirements and is delivered at *q* and *r* in the regular configuration { *p, q, r* }. However, *q* and *r* cannot deduce that *m*2

was received at *p*. Therefore, a transitional configuration { *q, r* } is delivered at both *q* and *r* followed by *m*2.

Message *m*3 which was sent by *p* was omitted by both *q* and *r* and was not recovered before the configuration change occurred. Hence, *m*4 is delivered at *q* and *r* immediately after *m*2. Although *m*5 was received by *q* and *r*, they **cannot** deliver it. *m*5 might be causally after *m*3 (which is true in this example) and does not meet the causal delivery requirement. Following that, the next regular configuration { *q, r, s, t* } is delivered at *q* and *r* so that they merge with *s* and *t*.

At *s* and *t*, all messages that were sent prior to the configuration change that merged them with *q* and *r* can meet the safe delivery requirements. Therefore all the messages that were sent in the regular configuration { *s, t* } such as *m*' are delivered in the regular configuration { *s, t* }, a transitional configuration { *s, t* } is delivered, and the next regular configuration { *q, r, s, t* } is delivered. Note that the transitional configuration is always empty when a merge occurs but no process from the old configuration partitions or crashes.

Notice that by delivering the transitional configuration, *q* and *r* comply with the agreed delivery requirements even though they cannot deliver *m3*. This is a major difference between extended virtual synchrony and virtual synchrony. Using the virtual synchrony, at least one of the two components { *q, r* } and { *p* } would have to block and loose its memory because of the potential inconsistency that occurs when *p* delivers *m3* at { *p, q, r*} while *q* and *r* do not. Extended virtual synchrony allows both components to continue, while providing them with useful information about the state of messages at both components.

4.3 Discussion

The Basic Delivery Specification 1.2, when restricted to a single configuration, expresses causality of events within a single processor.

While Specification 2.3 and Specification 2.4 require configuration change messages to define a consistent cut in the order of events at all the processors, processors are not required to recover messages sent in configurations they do not belong to. Specification 5 limits the causal delivery requirement to the same configuration, eliminating the need to recover the history of old configurations at other processors in order to meet causality.

Traditionally, definitions of causality include, in addition to Specification 5, a similar specification with $send_p(m,c)$ replaced by $deliver_q(m,c)$. Note that this new specification can be derived from the existing Specification 5 and Specification 1.3.

Specifications 5 through 7 represent increasing levels of service. Some systems may operate without the causal order requirement; other systems need the causal order requirement and may add a total order requirement and even a safe delivery requirement, as appropriate for the application.

Chapter 5

5. Group Communication Layer

The group communication layer provides reliable multicast and membership services according to the extended virtual synchrony model. We begin with a description of the Transis system that serves as our group communication layer. Next, we present the Ring reliable multicast protocol, one of the two reliable multicast protocols implemented in Transis. Lastly, we present some performance measurements of Transis using the Ring protocol. The Ring reliable multicast protocol [AMMAC93, AMMAC95] was developed and implemented by the author while the author visited the Totem project.

By presenting a relatively simple, yet highly efficient protocol that meets extended virtual synchrony, we show that extended virtual synchrony is indeed a practical model. Other protocols that meet this model exist in the Horus environment [vRBFHK95].

In this chapter the term “processor” is used to refer to an instance of the group communication layer running on a processor.

5.1 The Transis System

Transis is a group communication sub-system currently developed at The Hebrew University of Jerusalem. Transis supports the *process group* paradigm in which processes can join groups and multicast messages to groups. Using Transis, messages are addressed to the entire process group by specifying the group name (a string selected by the user). The group membership can change when a new process joins or leaves the group, when a processor containing processes belonging to the group crashes, or when a network partition or re-merge occurs. Processes belonging to the group receive configuration change notification when such an event occurs. The semantics of message delivery and of group configuration changes is strictly defined according to the extended virtual synchrony model.

Each processor that may have processes participating in group communication has one Transis daemon running. As can be seen in Figure 5.1 all the physical communication is handled by the Transis daemon. Each Transis daemon keeps track of processes residing in its processor, and participating in group communication. The Transis daemons keep track of the processors’ membership. This structure is in contrast to other group communication mechanisms where the basic participant is the process rather than the processor, and the group communication mechanisms are implemented as a library linked with the application process.

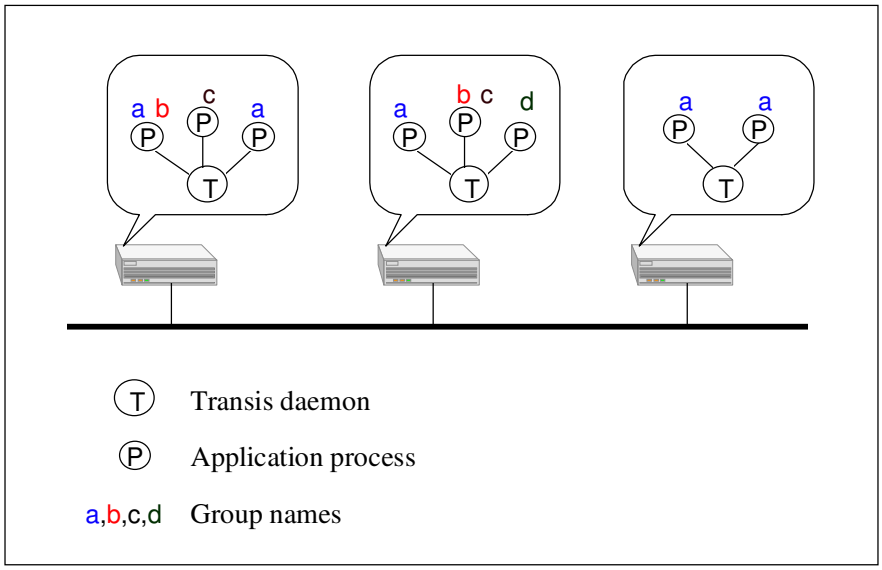

Figure 5.1: Process Groups in Transis

The benefits of this structure are significant:

- The membership algorithm is invoked only if there is a change in the processors' membership. When a process voluntarily joins or leaves a group, the Transis daemon sends a notification message to the other daemons. When this message is ordered, the daemons deliver a membership change message containing the new group membership to the other members of the group.
- Flow control is maintained at the level of the daemons rather than at the level of the individual process group. This leads to better overall performance.
- Order is maintained at the level of the daemons and not on a per group basis. Therefore, message ordering is more efficient in terms of latency and excessive messages.
- Message ordering across groups is trivial since only one global order, at the processors level, is maintained.
- Implementing open groups is easy (i.e. processes that are not members of a group can multicast messages to this group).

However, when necessary, the Transis daemon's code can be linked together with the user program, to create one process. This may be useful when a single program is using Transis and it is desirable to avoid the overhead of the inter-process communication.

Transis application programming interface (API) contains the following entries:

- **connect** - A process initiates a connection to Transis. This creates an inter process communication handle at the user process similar to a socket handle. A process can maintain multiple connections to Transis.
- **disconnect** - A process terminates a connection.

- **join** - A process voluntarily joins a specific process group on a connection. The first message on the group will be a membership notification of the currently connected members of the group.
- **leave** - A process voluntarily leaves a process group on a specific connection.
- **multicast** - A process generates a message to be multicast by Transis to a set of target groups. The order level required for delivery (causal, agreed, or safe delivery) is specified .
- **receive** - A process receives a message delivered by Transis on a specific connection. The message can be a regular message sent by a process, or a membership notification created by Transis regarding a membership change of one of the groups this process belongs to.

The Ring version of Transis is operational for almost three years now. It is used by students of the distributed systems course at the Hebrew University and by the members of the High Availability lab. Several projects were implemented on top of Transis, among them a highly available mail system, a distributed system management tool, and several graphical demonstration programs.

5.2 The Ring Reliable Multicast Protocol

The Ring reliable multicast protocol provides message delivery and membership services according to the extended virtual synchrony model. The protocol assumes the existence of a non-reliable multicast service in the system. Most local area networks have the ability to multicast or to broadcast messages. In systems with no multicast service, it can be mimicked by unreliable point-to-point message transmissions (unicast) without affecting the correctness of the protocol.

The Ring protocol carefully tailors three algorithms:

- **Message Ordering** - responsible for reliable and ordered delivery of messages. This algorithm handle message omissions.
- **Membership State Machine** - handles processor crashes and recoveries as well as network partitions and re-merges.
- **Extended Virtual Synchrony** - this algorithm is invoked after a membership change was detected and its members have been determined. It guarantees delivery of messages sent in the old configuration so that extended virtual synchrony is preserved.

The basic idea behind the ordering algorithm is **not** original work of the author. It was published in [MMA91]. The membership state machine and the algorithm for achieving extended virtual synchrony **are** original work of the author. The first part of the membership algorithm is based on the Transis membership algorithm [ADKM92b].

5.2.1 Message Ordering

The main principle of this algorithm is to achieve message ordering by circulating a token around a logical ring imposed on the processors (GC members) participating in the current configuration. Only the processor in possession of the token can multicast messages to the other members on the ring. Here we assume no token loss and no membership changes such as processor crashes and recoveries, or network partitions and re-merges. This cases are handled by the membership algorithm described in the next sub-section.

Message ordering is achieved by using a single sequence of message sequence numbers for all processors on the ring and by including the sequence number of the last message multicast in the token. Stamping each message with the sequence number places a total order on the messages before they are sent. Thus, each processor receiving the message can immediately determine its order.

Message Structure

Regular message contains the following fields:

- *type* - regular message.
- *conf_id* - a unique identifier of the configuration within which the message was multicast.
- *proc_id* - a unique identifier of the processor that multicast the message.
- *seq* - the sequence number of the message. This field determines the agreed order of the message.
- *data* - the content of the message.

The regular token contains the following fields:

- *type* - regular token.
- *conf_id* - a unique identifier of the configuration within which the token was multicast.
- *seq* - the highest sequence number of any message that has been multicast within this configuration. At the beginning of each regular configuration, the *seq* is set to zero.
- *aru* - A sequence number (all-received-up-to) such that all processors on the ring have received all messages up to and including the message with this sequence number. This field is used to provide safe delivery and to control the discarding of messages that have been received by all processors on the ring and that will, therefore, not need to be retransmitted. At the beginning of each regular configuration, the *aru* is set to zero.

- *rtr* - A retransmission request list, containing one or more retransmission requests. Each request contains (*conf_id, seq*) of the requested message.
- *fcc* - The number (flow control count) of messages actually multicast by all processors on the ring in the last rotation of the token, including retransmissions.

Message Multicast and Delivery

Each processor maintains a local variable *my_aru* containing the sequence number of the message such that it has received all messages with sequence numbers at most equal to that sequence number. At the beginning of each regular configuration, *my_aru* is set to zero. As the processor receives messages, it updates *my_aru*. Each processor maintains a list of messages that it has received; messages that are safe can be discarded from this list.

On receipt of the token, the processor multicast messages, updates the token and transmits it (unicast) to the next processor on the ring. For each new message it multicasts, the processor increments the *seq* field of the token and sets the sequence number of the new messages to this *seq*.

Whether multicasting a message or not, the processor compares the *aru* field of the token with *my_aru* and, if *my_aru* is smaller, it sets *aru* to *my_aru*. If the processor previously lowered the *aru* and the token returned with the same value, then it sets *aru* equal to *my_aru*. If *seq* and *aru* are equal, then it increments *aru* and *my_aru* in step with *seq*.

If the *seq* field of the token indicates that messages have been multicast that the processor has not yet received, the processor augments the list to the *rtr* field. If the processor has messages that appear in the *rtr* field then, for each such message, it generates an independent random variable to determine whether it should retransmit that message before multicasting new messages (this randomization increases overall system reliability). When it retransmits a message, the processor removes it from the *rtr* filed.

The *fcc* field provides the data needed for the flow control of the protocol as described in the performance section.

Message delivery is done as follows: If a processor has delivered every message with sequence number less than that of an agreed message *m*, then it can delivered *m* in agreed order. If a processor has delivered every message with sequence number less than that of a safe message *m*, and if on two successive rotations of the token it releases it with an *aru* no less than the sequence number of *m*, then it can deliver *m* in safe order.

5.2.2 Membership State Machine

The membership algorithm presented here is used in conjunction with the message ordering algorithm and with the extended virtual synchrony algorithm. The algorithm handles all aspects of processor membership, including processor failure and restart, token loss, and network partitioning and re-merging. The algorithm uses a single representative for each ring that is being merged; this representative negotiates the membership of the new ring on behalf of the other processors on the old ring and should not be regarded as a leader or master of the old or new rings. While a new ring is being formed, the old ring is used as long as possible to multicast new messages. Before installing the new regular configuration, the new ring is used to recover messages from the old configuration that must be delivered in order to achieve extended virtual synchrony.

The membership algorithm is defined in terms of the state diagram shown in Figure 5.2. The message structure and the definition of events and states are given below.

Message Structure

The membership algorithm uses two types of special messages which have no sequence numbers and are not delivered to the application:

- **Attempt Join** message multicast by a representative initiating the membership algorithm to form a new ring from two or more rings.
- **Join** message multicast by a representative proposing a set of representatives of old rings and also a set of failed representatives (a subset of the set of representatives). The proposed new ring will be formed by the representatives in the first set but not in the second.

In forming a new ring, the representative of that ring generates a **Form** token. The Form token which differs from the regular token of the message ordering algorithm, contains the following fields:

- *type* - Form token.
- *form_id* - The Form token identifier, which consists of the identifier of the representative of the new ring and a timestamp representing the time of creation of this Form token. The *form_id* becomes the *conf_id* of the regular token once the regular configuration is installed.
- *join_list* - A sorted list of the representatives' identifiers.
- *memb_list* - A list containing all the identifiers of all the members of the new ring according to their position on the new ring. For each of these members, this list also contains its old configuration (*conf_id*) identifier.

- *confs_list* - A list containing a record for each (old) configuration that has a member participating in the new ring. This field is used by the extended virtual synchrony algorithm and is detailed there.

Definition of Events

There are five membership events, namely:

- **Receiving a *foreign* message**. The message can be one of:
  - ⇒ Regular message multicast by a processor that is not a member of the ring.
  - ⇒ Attempt Join message.
  - ⇒ Join Message.
- **Receiving a Form token**. On the first receipt of the Form token a processor of the proposed new ring updates the Form token; on the second receipt it obtains the updated information that the other processors supplied.
- **Token loss timeout**. This timeout indicates that a processor did not receive either the token or a message from some other processor on the ring within the timeout period.
- **Gather timeout**. This timeout is used to bound the time for gathering representatives to form a new ring.
- **Commit timeout**. This timeout indicates that a processor participating in the formation of a new ring failed to determine that an agreement had been reached on the members of the new ring.

Definition of States

There are five states, namely:

- **Operational state**. This is the regular state of the Ring protocol in which the message ordering algorithm operates with no membership changes.
- **Gather state**. The representatives that will constitute the new ring are collected. This is done by gathering as many Attempt Join and Join messages as possible before the Gather timeout expires.
- **Commit state**. The representatives attempt to reach agreement on the set of representatives whose rings will be combined to form the proposed new ring.
- **Form state**. The path of the token of the proposed new ring is determined and information about the members of that ring is exchanged.
- **EVS state**. The extended virtual synchrony algorithm is invoked in this state of the Ring protocol in order to guarantee the requirements of the extended virtual synchrony model.

Formation of a New Ring

We first explain the membership algorithm without considering the effects of further processor failure or token loss during operation of the algorithm. Those effects are examined in the next sub-section.

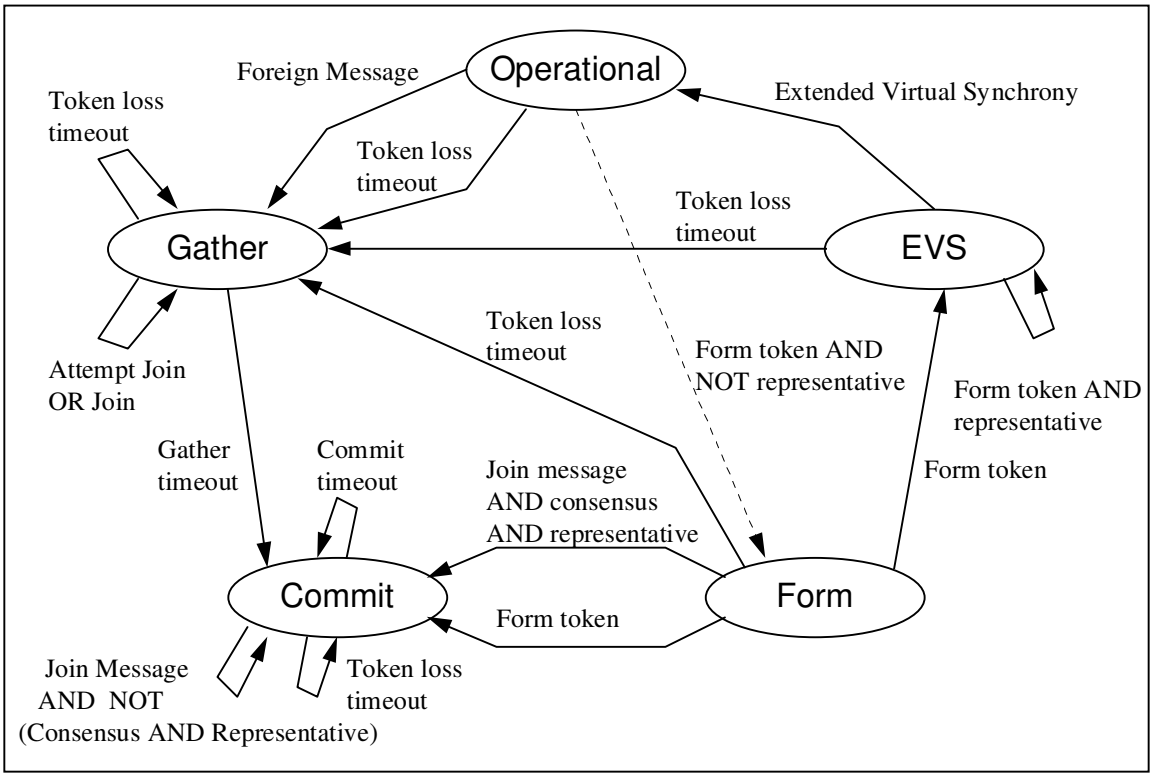

Figure 5.2: The State Diagram for the Membership Algorithm

The membership algorithm is invoked when a token loss is detected or when a foreign message is received by a processor on the ring. A processor that recovers forms a singleton configuration containing only itself and immediately shifts to the Gather state. A non-representative in the Operational state ignores foreign messages.

In the Operational state the ordering of messages proceeds according to the message ordering algorithm of the Ring protocol. When a foreign message is received, the representative multicasts an Attempt Join message, advertising its intention to form a bigger ring. It then shifts to the Gather state.

The Gather state allows time for the representative to collect together as many representatives as possible to form a new ring. The representative remains in the Gather state until the Gather timeout expires. It then multicasts a Join message, containing the identifiers of the representatives it has collected in the Gather state, and shifts to the Commit state.

In the Commit state the representatives reach an agreement. They agree on the set of representatives that will participate in the formation of the new ring. In order to reach

agreement, each representative multicasts a Join message containing a set of representatives and a set of failed representatives. An agreement is reached when there exists a set of representatives and a set of failed representatives, listed in a Join message, such that each of the non-failed representatives has multicast a Join message with exactly these two sets. In the Commit state, the two sets of a representative are not decreasing. A representative which sent a Join message, cannot multicast a different message unless another representative, needed for the agreement, multicasts a Join message which contains any representative, in either of the two sets, that is not included in the sets of the first Join message. In this case, the second representative will never agree on the sets in the first Join message. Therefore, the first representative must multicast a new Join message containing the union of the both sets in the two Join messages.

If the Commit timeout expires before agreement has been reached then the representative inserts all representatives from which it has not received the required Join message into the set of failed representatives. It then multicasts a revised Join message, restarts the Commit timeout, and tries again to form a new ring.

The representative for the proposed new ring, chosen deterministically from among the representatives when an agreement is reached, generates a Form token. The Form token circulates through all members of the proposed new ring along a cycle determined by the increasing order of identifiers of the representatives (see Figure 5.3). Every member of the proposed new ring shifts to the Form state as it forwards the Form token. This includes the non-representatives, which shift from the Operational state to the Form state, as shown by the dashed line in Figure 5.2. Having entered the Form state, a processor consumes the regular token for the old ring if it subsequently receives it.

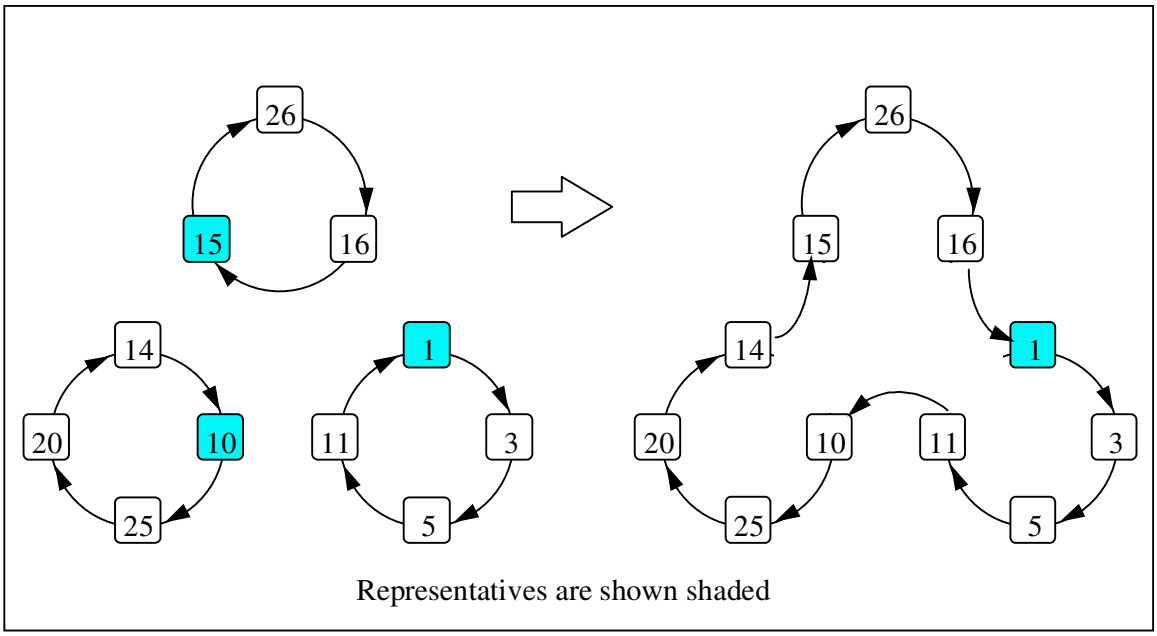

Figure 5.3: The Token Path After a Merge.

After one rotation of the Form token, the representative of the new ring knows all the information needed for the extended virtual synchrony algorithm and for installing the new ring, and shifts to the EVS state. After the second rotation of the Form token, all other processors on the proposed new ring have this information and have also shifted to the EVS state. On receiving the Form token after its second rotation, the representative of the new ring consumes the Form token and transmits the regular token for the new ring in its place. At this point the new ring is formed, but neither the transitional configuration nor the new regular configuration are installed (delivered). When shifting to the EVS state, the extended virtual synchrony algorithm is invoked.

Token Loss, Processor Failure and Network Partition

The algorithm does not distinguish between processor failure and token loss because a failed processor cannot forward the token to the next processor on the ring. Thus, the consequence of processor failure is a token loss. Network partition has similar effects. If the token reaches a processor that has (mistakenly) determined that the token is lost, the processor consumes the token.

The most common token loss event occurs in the Operational state. On expiration of the token loss timeout, a processor regards itself as a representative, representing only itself but retaining its existing *conf_id*, and proceeds to the Gather state.

Token loss can also occur to a representative in the Gather or Commit states. If, in either of these states, the token of the representative's existing ring is lost, the representative continues the operation of the membership algorithm, retaining its existing *conf_id* but representing only itself.

Loss of the Form token can also occur in the Form state. In this state, the old ring is no longer operational and the new ring has not yet been formed, and the processor returns to the Gather state. The membership algorithm ensures termination by adding the processor with the highest identifier to the set of failed processors. The next membership on which agreement is reached at the Commit state cannot include this member.

A failure in the EVS state is handled by the extended virtual synchrony algorithm described below.

5.2.3 Achieving Extended Virtual Synchrony

The basic idea of the extended virtual synchrony algorithm is to use the newly created ring, and the regular message ordering algorithm to recover lost messages that were sent at the old regular configuration. The extended virtual synchrony algorithm is invoked while in the EVS state. While in this state, the processor does not send new messages. There is only one token circulating in the new ring. Retransmission requests and retransmitted messages are ignored by processors not belonging to the configuration

specified by the *conf_id* field both in the retransmission request and in the retransmitted message.

After extended virtual synchrony is reached at all the members of the new ring, each processor performs the following steps as an atomic action and installs the new configuration:

1. Deliver messages that can be delivered in the old regular configuration.
2. Deliver a transitional configuration.
3. Deliver messages that could not be delivered in the old regular configuration, but need to be delivered in order to meet self delivery, causal delivery, agreed delivery, and safe delivery requirements.
4. Deliver a regular configuration composed of the members of the new ring.

Data Structure

The *confs_list* in the Form token is used to gather information about messages that were sent at the old configurations. The *confs_list* contains a record for each (old) configuration that has members participating in the new ring. Each record contains the following fields:

- *conf_id* - the configuration identifier of the configuration related to this record.
- *obligation_set* - a subset of the processors which are member of the regular configuration. Processors in the new ring transitioning from this (old) configuration, deliver all the messages in that configuration that were originated by members of the *obligation_set*. This is done in order to satisfy the self delivery, causal delivery, and safe delivery requirements.
- *highest_seq* - the highest sequence number of a message that is known to be multicast in the (old) configuration.
- *aru* - the highest *aru* known in the (old) configuration.
- *holes* - a set of sequence numbers of messages from the old configuration that are missing by all the members of the old configuration participating in this new ring. All these numbers will be higher than the *aru* and lower or equal to the *highest_seq*.

Each processor maintains the following local variables:

- *my_obligation_set* - a subset of processors which are members of the (old) configuration. Initially, when a processor leaves the Operational state, the *obligation_set* contains only itself.
- *original_aru* - the highest reported *aru* by a member of the old configuration.
- *barrier_seq* - this sequence number is set to be one plus the highest *highest_seq* of all the records in *confs_list*.

The Extended Virtual Synchrony Algorithm

After reaching agreement among the members of the new ring, and circulating the Form token twice, all the members of the new ring share the same information regarding messages and members of old configurations participating in the new ring. In particular, the members of each specific old configuration participating in the new ring, agree upon the *highest_seq, aru*, and the set of *holes* for their old configuration. They set their local variables, set *original_aru* to *aru*, and create a place holder for each missing message.

At the second rotation of the Form token each member shifts to the EVS state. The representative of the new ring consumes the From token and initiates a regular token with *seq* initiated to zero, empty *rtr*, *aru* initiated to its *highest_seq*. The processor, then, operates according to the message ordering algorithm as if it was in the Operational state.

Note that, although a single token in used, retransmission requests from different old configurations can reside in this single token, and messages from different old configurations can be retransmitted. In the EVS state, each processor ignores foreign messages and foreign retransmission requests. Since the holes are filled with place holders, the *my_aru* of each member will be able to reach the corresponding *highest_seq*.

When *my_aru* of a processor reaches *highest_seq* this processor finished recovering all of the messages from the old configuration. It sets *my_aru* to be *barrier_seq*. Eventually, after all the processors on the new ring finish recovering, the *aru* reaches *barrier_seq*. At this point, within one rotation of the token, each of the processors performs the following steps and shifts to the Operational state:

1. Deliver in order all of the messages up to but not including the first place holder, or the first safe message higher than *original_aru*.
2. Deliver a transitional configuration that includes all the members of the old configuration participating in this new ring.
3. Discard (without delivering) all the messages, except those sent by a member of the *obligation_set*. Such messages must be discarded because they may be causally dependent on an unavailable message. In addition, all the place holders are discarded. Note that the *obligation_set* includes (at least) all the members of the transitional configuration.
4. Deliver in order all the remaining messages.
5. Deliver a regular configuration that includes all the members of the new ring.

Steps 1-5 are performed locally as an atomic action without communication with any other process.

Failure at the EVS state

If the token was lost at this state the processor returns to the Gather state after performing the following steps:

1. If its *my_aru* is equal to *barrier_seq* this processor finished recovering all of the messages from the old configuration. It also promised to the other members that it will deliver messages according to the *obligation_set*. Therefore, the processor sets *my_obligation_set* to *obligation_set*. This protects processors that might have installed the new configuration in case the token was lost at the installation round.
2. Discard the place holders.
3. Sets its *aru* back to *original_aru*, and sets *my_aru* back to the sequence of the highest consecutive message it has.
4. Places the processor with the highest identifier, not including itself, in the set of failed processors in order to ensure termination.

The processor then tries again to reach agreement on the membership, to form a new ring, to achieve extended virtual synchrony, and to install a new configuration.

5.3 Performance

The Ring reliable multicast protocol constitutes one of the two reliable multicast protocols of Transis. The implementation, written in C, uses the standard non-reliable UDP/IP interface. within the Unix operating system. The Transis code compiles and runs on several types of Unix machines including Sun Sparcs, Silicon Graphics machines, IBM R-6000, and the 386 architecture with either Netbsd, Linux or BSDI Unix.

We used 16 Pentium PC machines running BSDI Unix for the following experiments. The machines are connected by a single 10 megabits per second Ethernet segment. The Ethernet interface card used is SMC 8216 LAN adapter. There is almost no external load on the machines.

Although performance measured on other architectures (Sparcs, SGIs) gives similar or sometimes even slightly better results, we have decided to conduct our experiments on the cheapest, as well as the most popular, architecture.

Figure 5.4 presents the maximal throughput of the protocol using 1Kbyte messages, equally shared among the processors, as a function of the number of processors participating in the ring. The average throughput measured is 860 messages per second, with a maximum throughput of 872 messages per second (with 4 processors) and a minimum throughput of 852 message per second (with 6 and 15 processors). 856 messages per second were measured with 16 processors. Apparently, the throughput measured is almost not affected by the number of processors on the ring.

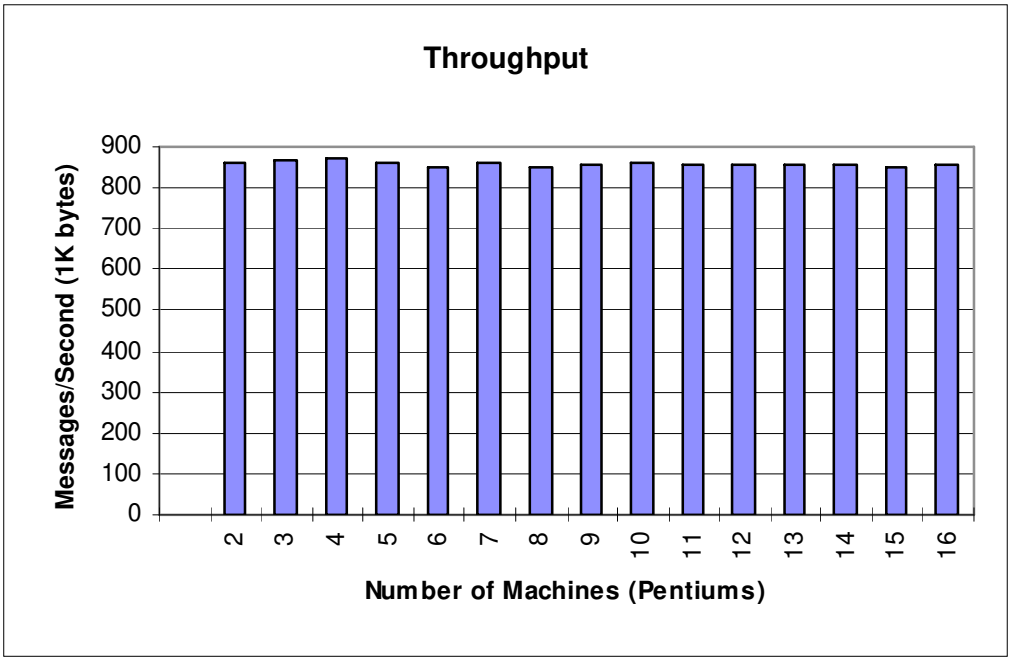

Figure 5.4: Throughput as a Function of the Number of Machines

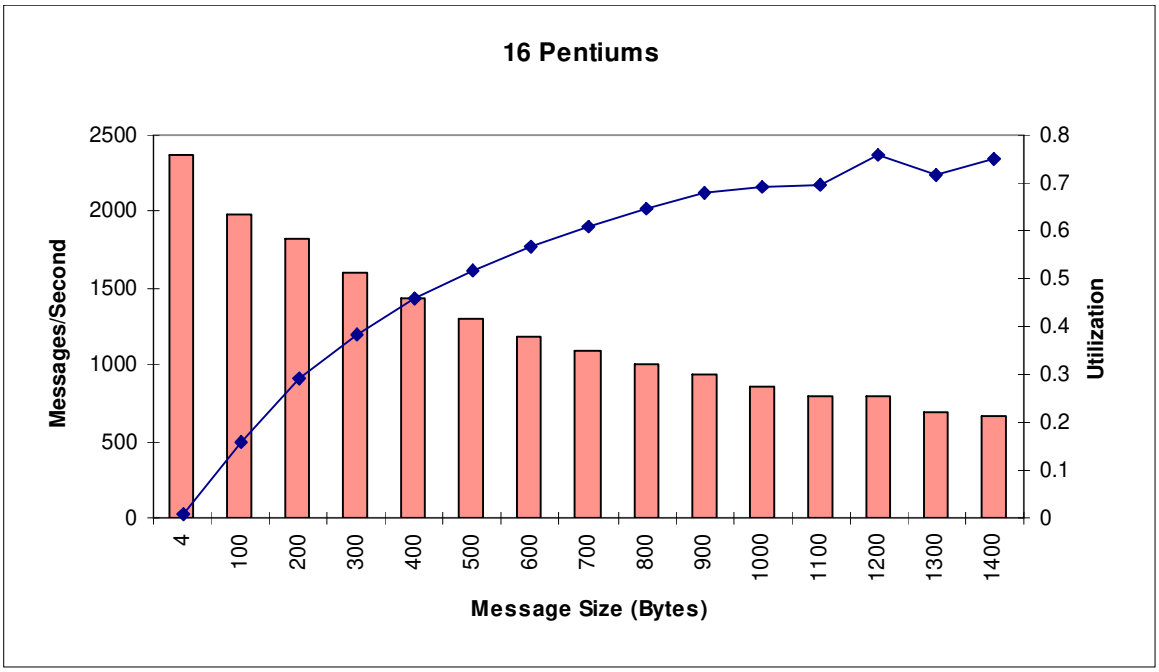

Figure 5.5: Throughput as a Function of Message Size

These measurements were taken when Transis flow control was tuned for best performance, allowing each processor to transmit 20 messages on each visit of the token. Hence, the number of messages (including retransmissions) transmitted in each token rotation were linearly increasing with the number of participating processors.

A concern about token-passing protocols is that the token-passing overhead reduces the transmission rate available for messages. Figure 5.5 depicts the useful utilization (excluding transmissions of the token, message headers, and retransmissions) of the Ethernet achieved by the protocol with a 16 processors. Over 70% utilization is achieved for 1Kbyte messages. Larger messages achieve slightly over 75% utilization (670 messages per second of 1400 bytes of data per message). Figure 5.5 also presents the maximal transmission rates achieved. Over 1000 messages per second are achieved when message size is limited to 800 bytes of data (1984 messages per second for 100 byte messages). Note that each message represents a distinct send operation on the network.

The latency to safe delivery, measured from the time a message is generated to the time it is delivered is presented in Figure 5.6. The tradeoff between latency and throughput are measured for 4,8,12 and 16 processors when the load is shared equally among the processors. All messages are of 1K bytes data size. The measurements are conducted by controlling the number of messages each processor can transmit on each visit of the token. In this way, the overall throughput can be controlled up to the maximal throughput displayed in Figure 5.4.

The latency to safe delivery is approximately twice the token rotation time when the load is less than the maximal throughput. The latency to agreed delivery is slightly more than half the token rotation time. Further, when the load is equally shared, the latency increases linearly with the number of processors participating on the ring.

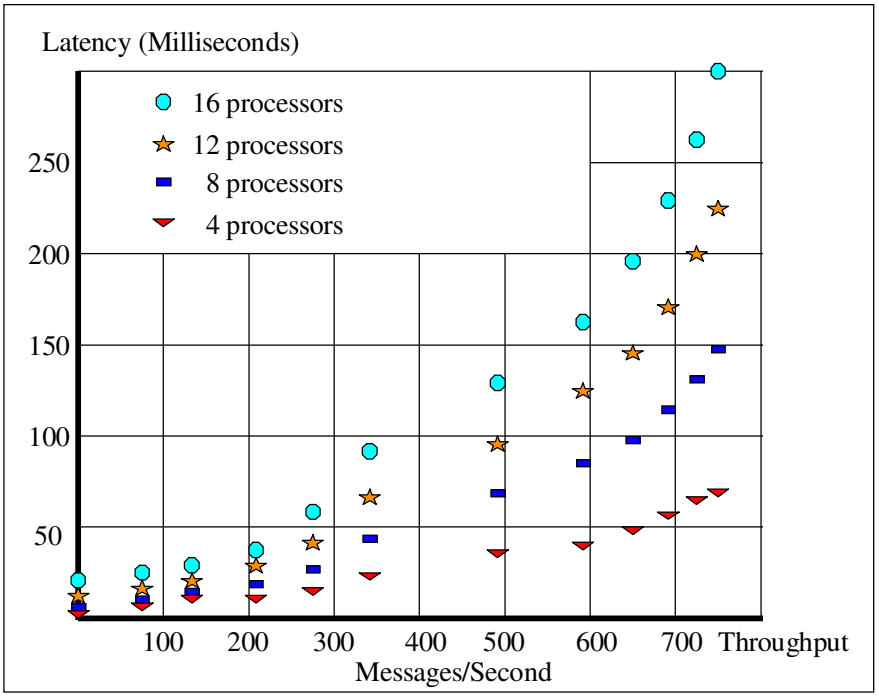

Figure 5.6: Latency to Safe Delivery as a Function of Throughput

Chapter 6

6. Replication layer

The replication layer implements a symmetric algorithm for guaranteed delivery and for *global* total ordering of actions. This layer guarantees that all actions will eventually reach all servers, and will be applied to the database in the same order at all the replication servers.

The replication layer uses a group communication layer that, according to the extended virtual synchrony model, maintains the membership of the currently connected servers and *locally* orders actions within this current membership. The task of creating a global total order out of the local order provided by the group communication is non-trivial due to the requirement to overcome network partitions and processor crashes. Moreover, our aim to allow all components of the partitioned network to continue operation, although in some degraded mode, adds additional complexity to the problem.

The most challenging aspect of the replication layer is its ability to globally order actions consistently **without** the need for end-to-end acknowledgment on a per-action basis between the replicas, and **without** loosing actions in case of processor crashes (and power failures). Other consistent replication mechanisms that tolerate the same failure model require every replica to perform a synchronous disk write per action, before they send an acknowledgment, and before this action can be applied to the database at other servers. Our unique property is achieved using the additional level of knowledge provided by the strong (yet relatively cheap) safe delivery property of the extended virtual synchrony model.

Not all applications require global total order of actions. Some applications may not care about the order of actions. Refer to Chapter 7 for optimized services for different types of applications.

6.1 The Concept

Since the servers group may partition, the replication layer identifies at most a single component of the servers group as a *primary component*; the other components of the partitioned servers group are *non-primary components*. Only the primary component determines the global order of actions. Servers belonging to non-primary components can still generate actions, but cannot determine their global order. According to the extended virtual synchrony model, a change in the membership of a component of the servers group

is reflected in the delivery of a configuration change message by the group communication layer to each server in that component that did not crash.

We use the following coloring model to indicate the knowledge level associated with each action. Each server marks the actions delivered by the group communication layer with one of the following colors:

- **Red Action**. An action for which the server cannot, as yet, determine the global order.
- **Green Action**. An action for which the server has determined the global order and which, therefore, can be applied to the database.
- **White Action**. An action for which the server can deduce that all of the servers in the servers group have already marked the action green. Thus, the server can discard a white action because no other server will need this action subsequently.

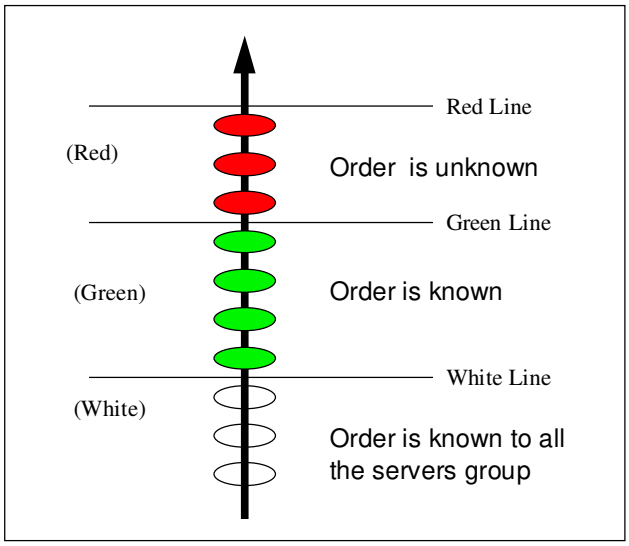

Figure 6.1: The Actions Queue at Server *s*.

All of the white actions precede the red and green actions in the global order and define the white zone. All of the green actions precede the red actions in the global order and define the green zone. Similarly, the red actions define the red zone. An action can be marked at different servers with different colors. However, the algorithm of the replication server guarantees that no action can be marked white at one server while it is marked red, or does not exist, at another server. A similar coloring model with a slightly different meaning appears is [AAD93,Kei94].

6.1.1 Conceptual Algorithm

We now present a high-level description of the algorithm in the form of a finite state machine with four states, as shown in Figure 6.1:

- **Prim state**. A server currently belongs to the primary component. When a message containing an action is delivered by the group communication layer, the action is **immediately** marked green and is applied to the database.
- **Non_prim state**. A server belongs to a non-primary component. When a message containing an action is delivered by the group communication layer, the action is **immediately** marked red.
- **Exchange state**. A server shifts to this state when a new (regular) configuration is formed. All of the servers belonging to the new configuration exchange information that allows them to define the set of actions that are known to some, but not all, of them. After all of these actions have been exchanged and the green actions have been applied to the local database, the server checks whether this configuration can form the next primary component. If so, it shifts to the Construct state; otherwise, it shifts to the Non_prim state and forms a non-primary component. We use dynamic linear voting [JM90] to determine if the next primary component can be formed. This check is done locally at each server without the need for additional exchange of messages among the servers.
- **Construct state**. In this state, all of the servers in the component have the same set of actions and know about the same set of former primary components. After writing the data to stable storage, the server multicasts a Create Primary Component (CPC) message. On receiving a CPC message from each of the other servers in the current configuration, a server shifts to the Prim state. If a configuration change occurs before it has received all of the CPC messages, the server returns to the Exchange state.

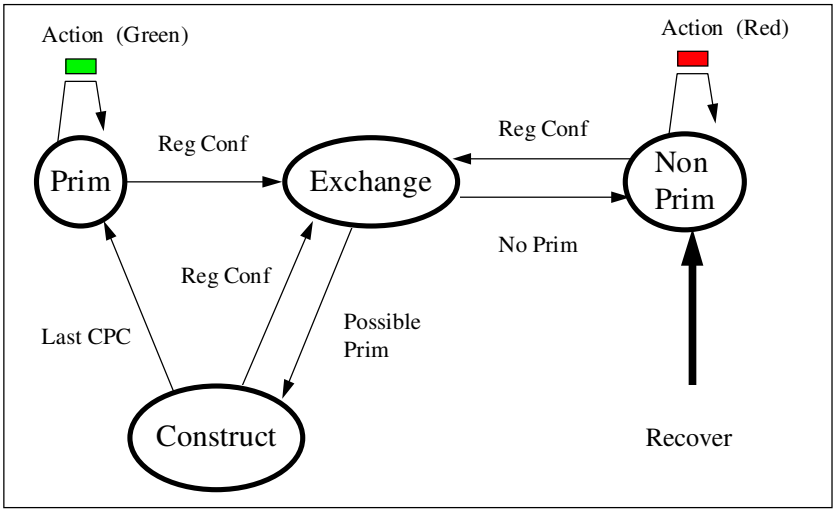

Figure 6.2: A Conceptual State Machine of the Replication Server

When a membership change occurs, the connected servers exchange information and try to reach a common state. If another membership change occurs before the servers establish that common state, they try again. When they reach a common state, that state will be either a Prim state or a Non_prim state.

In the general case, the server resides either in the Prim state or in the Non_prim state. When the servers within a component are in these two states, there is **no** need to acknowledge messages. As long as no membership change occurs, all of the connected servers receive the same set of messages in the same order, and there is no need for end-to-end acknowledgments. We still need end-to-end acknowledgments (i.e. server to server) after a membership change, but we avoid end-to-end acknowledgment on a per action basis. This means that, in the primary component, the replication servers can globally order messages without incurring additional delay to the latency of the group communication layer. Refer to Chapter 5 for latency measurements.

6.1.2 Selecting a Primary Component

In a system that is subject to partitioning, we must ensure that two different components do not reach contradictory decisions regarding the global order. Hence, we need a mechanism for selecting the primary component that can continue to order actions. Several techniques have been described in the literature [Gif79, JM90, Tho79]:

- **Monarchy**. The component that contains a designated server becomes the primary component.
- **Majority**. The component that contains a (weighted) majority of the servers becomes the primary component.
- **Dynamic Linear Voting**. The component that contains a (weighted) majority of the last primary component becomes the primary component. Sometimes, a lower bound is imposed on the size (weight) of the primary component to avoid situations where a crash of a small number of machines that formed the last primary component blocks the whole system.

Dynamic linear voting is generally accepted as the best technique, when certain reasonable conditions hold [PL88]. The choice of the weights and adapting them over time is beyond the scope of this thesis. We employ dynamic linear voting.

Any system that employs (weighted) dynamic linear voting can use (weighted) majority, since majority is a special case of dynamic linear voting. Monarchy is a special case of weighted majority (when all servers except the master have weight zero). However, it is not always easy to adapt systems that work well for monarchy or majority to dynamic linear voting.

6.1.3 Propagation by Eventual Path

In many systems, processes exchange information only as long as they have a direct and continuous connection. In contrast, the concept described above propagates information by means of eventual path.

An *eventual path* from server *s* to server *r* is a path from *s* to *r* such that there exist pairs of servers along the path and intervals during which they are connected so that information known to *s* is made known to *r* during this interval. This does **not** require a continuous or direct connection between *s* and *r*.

According to our concept, when a new component is formed, the servers exchange knowledge in the Exchange state. When servers leave the Exchange state, they share the same knowledge regarding the actions in the Action list and their order and color. Our method for sharing this information is efficient because the exchange process is invoked immediately after a configuration change, and only then. Moreover, each needed action is multicast exactly once using our well performing group communication layer.

Our concept might be compared with former point-to-point gossip and epidemic replication methods [LLSG92, Gol92]. In these methods, each server exchanges information from time to time with some connected server. Although these methods also meet the liveness criterion described in Chapter 2, for the above reasons, our method is more eager and disseminates the knowledge, in principle, immediately when communication resumes, using multicast. The reason this behavior can be achieved is that we exploit group communication multicast and membership services.

6.2 The Algorithm

Due to the asynchronous nature of the system model, we cannot reach complete knowledge about which actions were delivered to which servers just before a network partition or processor crash occurs. In fact, it is well known that reaching agreement in asynchronous environments with a possibility of even one failure is impossible [FLP85]. Instead, we rely on the extended virtual synchrony semantics for safe delivery, particularly when a safe message is delivered in a smaller transitional configuration. The lack of complete knowledge is evident when:

- A server is in the Prim state when a partition occurs. The server cannot always deduce whether the last actions were delivered to all the members of the primary component (including itself).
- A server is in the Construct state when a partition occurs. The server cannot always deduce whether all the servers in the proposed primary component have initiated the CPC messages.

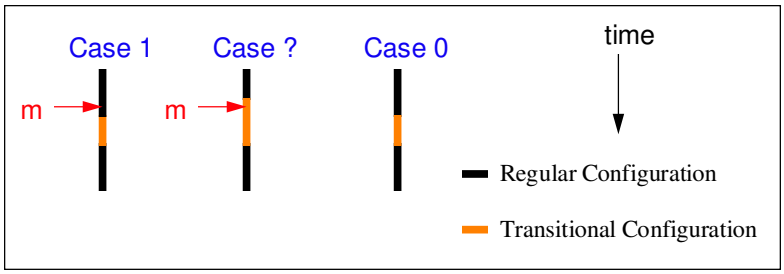

Figure 6.3 Three Cases to Break Impossibility

Extended virtual synchrony and its safe delivery property provides a valuable tool to deal with this incomplete knowledge. Instead of having to decide on one of two possible values (0 or 1) as in the consensus problem [FLP85], we have three possible values (0, ?, or 1) As Figure 6.3 presents:

- **Case 1**. A safe message is delivered in the regular configuration.
- **Case ?**. A safe message is received by the group communication layer just before a partition occurs. The group communication layer cannot tell whether other components that split from the previous component received and will deliver this message. According to extended virtual synchrony this message is delivered in the transitional configuration.
- **Case 0**. A safe message is sent just before a partition occurs, but it was not received by the group communication layer in a detached component. This message will not be delivered at this component.

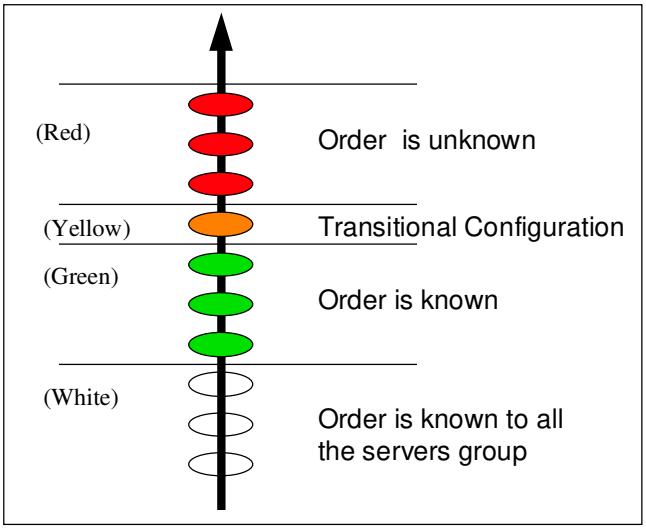

Figure 6.4: The Modified Color Model

In order to handle this uncertainty we modify the algorithm's state machine. We split the Prim state into the Reg_prim and Trans_prim states, and we add the No state (no server has yet installed the new primary component) and Un state (it is unknown whether any server has installed this component) as refinements of the Construct state. We add an intermediate color to our coloring model:

- **Yellow Action**. An action that was delivered in a transitional configuration of a primary component.

We mark as yellow actions that were delivered in a transitional configuration of a primary component. Such actions could have been marked as green by another member of a primary component that partitioned. A yellow action becomes green at a server as soon as this server learns that another server marked it green, or with the installation of the next primary component. The modified color model is presented in Figure 6.4 and the detailed state machine is presented in Figure 6.5.

Data Structure

The structure *Action_id* contains two fields: *server_id* the creating server identifier, and *action_index*, the index of the action created at that server.

The following local variables reside at each of the replication servers:

- *Server_id* - a unique identifier of this server in the servers group.
- *Action_index* - the index of the next action created at this server. Each created action is stamped with the Action_index after it is incremented.
- *Conf* - the current configuration of servers delivered by the group communication layer. Contains the following fields:
  - ⇒ *conf_id* - identifier of the configuration.
  - ⇒ *set* - the membership of the current connected servers.
- *Attempt_index* - the index of the last attempt to form a primary component.
- *Prim_component* - the last primary component known to this server. It contains the following fields:
  - ⇒ *prim_index* - the index of the last primary component installed.
  - ⇒ *attempt_index*- the index of attempt by which the last primary component was installed.
  - ⇒ *servers* - identifiers of participating servers in the last primary component.
- *State* - the state of the algorithm. One of {Reg_prim, Trans_prim, Exchange_states, Exchange_actions, Construct, No, Un, Non_prim}.
- *Actions_queue* - ordered list of all the red, yellow and green actions. White actions can be discarded and, therefore, in a practical implementation, are not in the

Actions_queue. For the sake of easy proofs this thesis does not extract actions from the *Actions_queue*. Refer to [AAD93] for details concerning message discarding.

- *Ongoing_queue* - list of actions generated at the local server. Such actions that were delivered and written to disk can be discarded. This queue protects the server from loosing its own actions due to crashes (power failures).
- *Red_cut* - array[1..n] - the index of the last action server *i* has sent and that this server has.
- *Green_lines* - array[1..n] - identifier of the last action server *i* has marked green as far as this server knows. *Green_lines*[*Server_id*] represents this server's green line.
- *State_messages* - a list of State messages delivered for this configuration.
- *Vulnerable* - a record used to determine the status of the last installation attempt known to this server. It contains the following fields:
  - ⇒ *status* - one of {Invalid, Valid}.
  - ⇒ *prim_index* - index of the last primary component installed before this attempt was made.
  - ⇒ *attemp_index* - index of this attempt to install a new primary component.
  - ⇒ *set* - array of *server_id*s trying to install this new primary component.
  - ⇒ *bits* - array of bits, each of {Unset, Set}.
- *Yellow* - a record used to determine the yellow actions set. It contains the following fields:
  - ⇒ *status* - one of {Invalid, Valid}
  - ⇒ *set* - an ordered set of action identifiers that are marked yellow.

Message Structure

Three types of messages are created by the replication server:

- Action_message - a regular action message contains the following fields:
  - ⇒ *type* - type of the message. i.e. Action
  - ⇒ *action_id* - the identifier of this action.
  - ⇒ *green_line* - the identifier of the last action marked green at the creating server at the time of creation.
  - ⇒ *client* - the identifier of the client requesting this action.
  - ⇒ *query* - the query part of the action.
  - ⇒ *update* - the update part of the action.

- State_message - contains the following fields:
  - ⇒ *type* - type of the message. i.e. State
  - ⇒ *Server_id, Conf_id, Red_cut, Green_line* - the corresponding data structures at the creating server.
  - ⇒ *Attempt_index, Prim_component, Vulnerable, Yellow* - the corresponding data structures at the creating server.
- CPC_message - contains the following fields:
  - ⇒ *type* - type of the message.
  - ⇒ *Server_id, Conf_id* - the corresponding data structures at the creating server.

Definition of Events

Six types of events are handled by the replication server:

- **Action** - an action message was delivered by the group communication layer.
- **Reg_conf** - a regular configuration was delivered by the group communication layer.
- **Trans_conf** - a transitional configuration was delivered by the group communication layer.
- **State_mess** - a state message was delivered by the group communication layer.
- **CPC_mess** - a Create Primary Component message was delivered by the group communication layer.
- **Client_req** - a client request was received from a client.

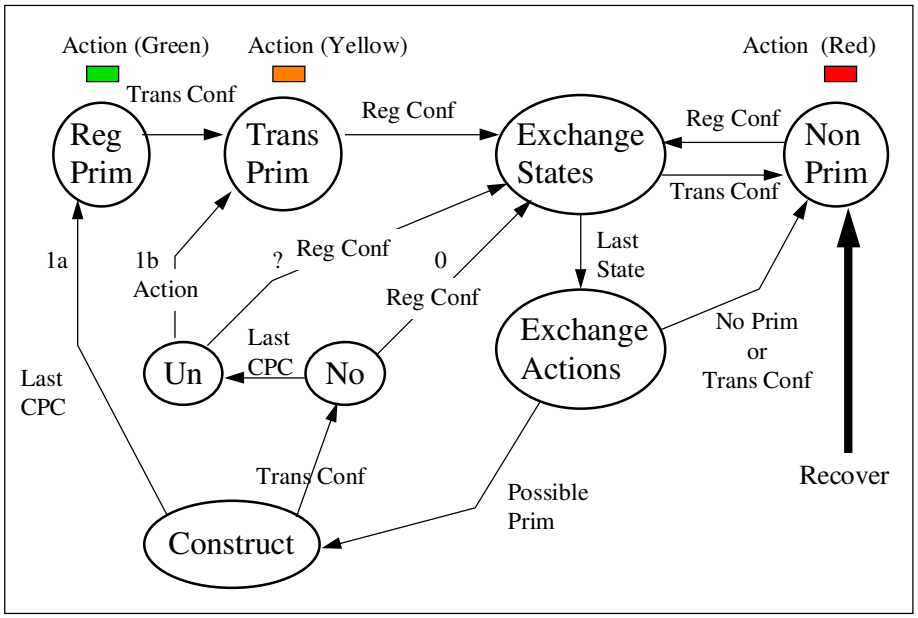

Figure 6.5: The State Machine of the Replication Server.

Non_prim State

While in the Non_prim state, each action is immediately marked as red. Client request generates an action that is sent by the group communication. This request is logged to disk in the *Ongoing_queue* in case the server crashes before this action is delivered and processed. The pseudo-code executed in the Non_prim state is presented in Figure 6.6.

```
case event is

  Action:
        Mark_red( Action )

  Reg_conf:
        set Conf according to Reg_conf
        Shift_to_exchange_states()

  Trans_conf, State_mess:
        Ignore

  Client_req:
        Action_index++
        create action and write to Ongoing_queue
        ** sync to disk
        generate Action

  CPC_mess:
        Not possible
```

Figure 6.6 Code Executed in the Non_prim State.

Reg_prim State

In this state, as soon as an action is delivered by the group communication layer, it is marked green. This is the most important property of the algorithm. There is no need to wait for other messages nor to write to disk. As long as it is in the Reg_prim state, the server is vulnerable. i.e. *Vulnerable.status* is valid so that if this server crashes, it will have to go through exchanging states and actions with another server belonging to this configuration before being able to participate in a primary component.

When a transitional configuration is delivered by the group communication layer, the server shifts to the Trans_prim state. The pseudo-code executed in the Reg_prim state is presented in Figure 6.7.

Trans_prim State

Actions delivered in the Trans_prim state are marked yellow. These actions might have been marked green at some partitioned server (e.g. message *m2* in Figure 4.8).

When a regular configuration is delivered by the group communication layer, the replication server knows that all the messages from the previous configuration were delivered and processed. This server is not vulnerable anymore, and its yellow set is valid because it received all the messages delivered by the group communication layer for the last configuration. The pseudo-code executed in the Trans_prim state is presented in Figure 6.8.

```
case event is

  Action:
        Mark_green( Action )                                    ( OR-1.1 )
        Green_lines[ Action.server_id ] = Action.green_line

  Trans_conf:
        State = Trans_prim

  Client_req:
        Action_index++
        create action and write to Ongoing_queue
        ** sync to disk
        generate Action

  Reg_conf, State_mess, CPC_mess:
        Not possible
```

Figure 6.7: Code Executed in the Reg_prim State.

```
case event is

  Action:
        Mark_yellow( Action )

  Reg_conf:
        set Conf according to Reg_conf
        Vulnerable.status = Invalid
        Yellow.status = Valid
        Shift_to_exchange_states()

  Client_req:
        buffer request

  Trans_conf, State_mess, CPC_mess:
        Not possible
```

Figure 6.8: Code Executed in the Trans_prim State.

Exchange_states State

In this state, the server gathers all the State messages sent by the currently connected servers. After receiving all these messages the server shifts to the Exchange_actions state.

The most updated server will be the first to retransmit. Just before shifting to the Exchange_action state, this server determines the last action ordered by all the currently connected servers and multicasts all **ordered** actions above that action, and then, all other actions according to a FIFO order. After doing that, all the servers have the same set of green actions in the *Actions_queue*.

Note that actions that were generated by a server just before a configuration change (but were not sent by the group communication before the configuration change) will be delivered before the State message of that server. These actions are marked red. The pseudo-code executed in the Exchange_states state is presented in Figure 6.9.

```
case event is

  Trans_conf:
        State = Non_prim

  State_mess:
        If (State_mess.conf_id = Conf.conf_id )
              add State_mess to State messages
        if ( all state messages were delivered )
              if ( most updated server ) Retrans()
              Shift_to_Exchange_actions()

  Action:
        Mark_red( Action )

  CPC_mess:
        Ignore

  Client_req:
        buffer request

  Reg_conf:
        Not possible
```

Figure 6.9: Code executed in the Exchange_states State.

The Shift_to_exchange_states procedure is invoked each time the server shifts to the Exchange_states state. The data structure is written to disk so that if the server crashes, the data reflected in the State message, which is sent by the server, will indeed be possessed by the server. The Shift_to_exchange_actions procedure is invoked when the server shifts to the Exchange_actions state. This procedure checks whether the retransmission process was completed in case no messages need to be retransmitted. The End_to_retrans procedure is invoked in the Exchange_actions state, just after all the

needed actions were retransmitted. Figure 6.10 presents the pseudo-code of Shift_to_exchange_states, Shift_to_exchange_actions, and End_of_retrans procedures.

```
Shift_to_exchange_states()

        ** sync to disk
        clear State_messages
        Generate State_mess
        State = Exchange_states


Shift_to_exchange_actions()

        State = Exchange_actions
        if ( end of retransmission )
                End_of_retrans()


End_of_retrnas()

        Incorporate all State_mess.green_line to Green_lines
        Compute_knowledge()
        if ( Is_quorum() )
                Attempt_index++
                Vulnerable.status = Valid
                Vulnerable.prim_index  = Prim_component.prim_index
                Vulnerable.attempt_index = Attempt_index
                Vulnerable.set = Conf.set
                Vulenrable.bits = all Unset

                ** sync to disk
                generate CPC message
                State = Construct
        else
                ** sync to disk
                Handle_buff_requests()
                State = Non_prim
```

Figure 6.10: Code for the Shift_to_exchange_states, Shift_to_exchange_actions, and End_of_retrans Procedures.

Exchange_actions State

In this state, the servers exchange actions that at least one of the servers has and one other server does not have. If a configuration change occurs and a regular configuration is delivered to the server, the server shifts back to the Exchange_states state.

After the most updated server finished retransmission, all the servers have the same set of green actions in the *Actions_queue* (see Exchage_states state). Other servers, one by

one, retransmit actions they have, that are needed to be retransmitted, and have not yet been retransmitted, according to a FIFO order.

Upon completion of retransmission, the server locally computes certain parameters in the data structure (refer to the End_of_retrans and Compute_knowledge procedures) and checks whether a new primary component can be formed. If the current component cannot form the next primary component, the server returns to the Non_prim state. Otherwise, the server marks itself vulnerable, writes its data structure to disk, multicasts a CPC message, and shifts to the Construct state. If the server subsequently crashes, it will know that an attempt to create a primary component was made with its participation. The pseudo-code executed in the Exchange_actions state is presented in Figure 6.11.

```
case event is

  Action:
        Mark action according to State_messages                    ( OR-3 )
        if ( turn to retransmit ) Retrans()
        if ( end of retrnasmission ) End_of_retrans()

  Trans_conf:
        State = Non_prim

  Client_req:
        buffer request

  Reg_conf, State_mess, CPC_mess:
        Not possible
```

Figure 6.11: Code Executed in the Exchange_actions State.

The Compute_knowledge procedure is invoked after retransmission is completed. The data structures at each of the connected servers are identical after the execution of this procedure. Step 1 of the procedure computes the most updated *Prim_component* and *Attempt_index* known to the connected servers. The servers with the most updated knowledge are identified in *Updated_group*.

Step 2 of the procedure computes the *Yellow* set based on the updated servers' valid sets. Messages that are contained in all of these sets are the only messages to be left on the *Yellow* set. Other messages are eliminated from the *Yellow* set because no server could have marked them green (otherwise, every updated server with a valid *Yellow* set will have them at least as yellow). If no such valid set exists, then the *Yellow* set is invalidated.

Steps 3 and 4 try to make vulnerable servers unvulnerable. In Step 3, a server that is vulnerable due to some old primary component or due to some old attempt to form a new primary component is marked unvulnerable. This is because the knowledge regarding this old primary component or old attempt exists, and is encapsulated in the *Action_list* and in the States messages. In Step 4, a case such as a total immediate crash of a primary component is handled. In this case, all the servers of that component are vulnerable to the

same set. Servers that realize this, mark themselves unvulnerable, because they now share all the information regarding the last primary component or the last attempt to form a primary component. Figure 6.12 presents the pseudo-code of the Compute_knowledge procedure.

```
Compute_knowledge()
1.      Prim_component = Prim_component in State_messages with
                        the maximal (prim_index, attempt_index)
        Updated_group = the servers that sent Prim_component in their State_mess
        Valid_group = the servers in Updated_group that sent Valid Yellow.status
        Attempt_index = max attempt_index sent by a server in Updated_group
                        in their State_mess

2.      if Valid_group is not empty
                Yellow.status = Valid
                Yellow.set = intersection of Yellow.set sent by Valid_group
        else
                Yellow.status = Invalid

3.      for each server with Valid in Vulnerable.status
                if ( server_id not in Prim_component.set
                    or
                    one of its Vulnerable.set does not have identical Vulnerable.status or
                        Vulnerable.prim_index or Vulanrable.attempt_index )
                then Invalid its Vulnerable.status

4.              for each server with Valid in Vulnerable.status
                set its Vulnerable.bits to union of Vulnerable.bits of
                        all servers with Valid in Vulnerable.status
                if all bits in its Vulnerable.bits are set then its Vulnerable.status = Invalid
```

Figure 6.12: Code of the Compute_knowledge Procedure.

```
Is_quorum()
        if there exist a server in Conf with Vulnerable.status = Valid return False
        if Conf does not contain a majority of Prim_component.set return False
        return True

Handle_buff_requests()
        for all buffered requests
                Action_index++
                create action and write to Ongoing_queue
        ** sync to disk
        for all buffered requests
                generate Action
        clear buffered requests
```

Figure 6.13: Code of the Is_quorum and Handle_buff_requests Procedures.

Construct State

In the Construct state, the server tries to gather all the CPC messages sent by the connected servers. Three possibilities exist: Either all the CPC messages are delivered, or a transitional configuration is delivered before all the CPC messages are delivered, or the server crashes.

If all the CPC messages are delivered, the server marks the yellow actions as green, installs the next primary component, and marks the red actions as green.

If a transitional configuration is delivered, the server shifts to the No state. Note that if the server crashes while in this state, it remains vulnerable when it recovers until it finds out how this installation attempt terminated. Refer to the description of the Compute_knowledge procedure for more information. The pseudo-code executed in the Construct state is presented in Figure 6.14.

```
case event is

  Trans_conf:
        State = No

  CPC_mess:
        if ( all CPC_mess were delivered )
              for each server s in Conf.set
                    set Green_lines[s] to Green_lines[ Server_id ]
              Install()
              State = Reg_prim
              Handle_buff_requests()

  Client_req:
        buffer request

  Action, Reg_conf, State_mess:
        Not possible
```

Figure 6.14: Code Executed in the Construct State.

The Install procedure marks green all the yellow actions and sets the *Prim_component* structure to reflect the installation. *Attempt_index* is set to zero to count the attempts to form the next primary component after this installed primary component breaks. The data structure is then written to disk so that the server remembers this installation even if it crashes. Figure 6.15 presents the pseudo-code of the Install procedure.

```
Install()
        if ( Yellow.status = Valid )
                for all actions in Yellow.set
                        Mark_green( Action )                    ( OR-1.2 )
        Yellow.status = Invalid
        Yellow.set = empty
        Prim_component.prim_index++
        Prim_component.attempt_index = Attempt_index
        Prim_component.servers = Vulnerable.set
        Attempt_index = 0
        for all red actions ordered by Action.Action_id
                Mark_green( Action )                            ( OR-2 )
        ** sync to disk
```

Figure 6.15: Code of the Install Procedure.

No State

The server reaches this state from the Construct state when a transitional configuration is delivered before all the CPC messages are delivered. Again, three possibilities exist: either all the CPC messages are delivered, or a regular configuration is delivered, or the server crashes.

If all the CPC messages are delivered, the server shifts to the Un state.

If a regular configuration is delivered, the server knows that no server received all the CPC messages while in the Construct state. Hence, the server marks itself unvulnerable and shifts to the Exchange_states state.

Note that if the server crashes while in this state, it remains vulnerable when it recovers, until it finds out how this installation attempt terminated. Refer to the description of the Compute_knowledge procedure for more information. The pseudo-code executed in the No state is presented in Figure 6.16.

Un State

The server in the Un state received all the CPC messages, although some of them were delivered in the transitional configuration. This server cannot tell whether there is a partitioned server that received all the CPC messages in the regular configuration (in the Construct state) and managed to install the primary component. Again three possibilities exist: either a regular configuration is delivered, or an action is delivered, or the server crashes.

If a regular configuration is delivered, the server must protect a potential partitioned server that installed the primary component. Therefore, the server remains vulnerable. This is the only place in the algorithm where the server is neither in the primary component, nor it is attempting to create a primary component, and it still remains vulnerable. However, the chances for this to happen are fairly slim and require a partition to occur exactly after all the CPC messages have been received, but before others acknowledge them.

If an action is delivered, the server knows that one of the servers installed the primary component (because only after that, an action can be sent). The server marks yellow actions as green, installs the primary component, and marks all red actions as green. Since the transitional configuration was already delivered, the server immediately shifts to the Trans_prim state. The delivered action is marked yellow because it was delivered in a transitional configuration of a primary component (the primary which was just installed).

Note that if the server crashes while in this state, it remains vulnerable when it recovers until it finds out how this installation attempt terminated. Refer to the description of the Compute_knowledge procedure for more information. The pseudo-code executed in the Un state is presented in Figure 6.17.

```
case event is

  Reg_conf:
        set Conf according to Reg_conf
        Vulnerable.status = Invalid
        Shift_to_exchange_states()

  CPC_mess:
        if ( all CPC_mess were delivered ) State = Un

  Client_req:
        buffer request

  Action, Trans_conf, State_mess:
        Not_possible
```

Figure 6.16: Code Executed in the No state.

```
case event is

  Reg_conf:
        set Conf according to Reg_conf
        Shift_to_exchange_states()

  Action:
        Install()
        Mark_yellow( Action )
        State = Trans_prim

  Client_req:
        buffer request

  Trans_conf, State_mess, CPC_mess:
        Not possible
```

Figure 6.17: Code Executed in the Un State.

Recover

When a processor recovers, it marks all actions residing in the *Ongoing_queue* and are not in *Actions_queue* as red. These actions were generated by this server, but were not delivered and processed by the server before it crashed. After cleaning its *Ongoing_queue,* the server shifts to the Non_prim state, waiting for the first regular configuration to be delivered. Figure 6.18 presents the pseudo-code of the Recover procedure.

```
Recover()
        State = Non_prim

        for each action in Ongoing_queue
                if ( Red_cut[ Server_id ] < Action.action_id.action_index )
                        Mark_red( Action )
        ** sync to disk
```

Figure 6.18: Code of the Recover Procedure.

Marking Actions

Three procedures mark actions: Mark_red, Mark_yellow and Mark_green. The first time an action is marked red, the Apply_red procedure is called. The first time an action is marked green, the Apply_green procedure is called. Figure 6.19 presents the pseudo-code of the marking procedures.

```
Mark_red( Action )

        if ( Red_cut[ Action.server_id ] = Action.action_id.index - 1 )
                Red_cut[ Action.server_id ]++
                Insert Action at top of Action_list
                if ( Action.type = Action ) Apply_red( Action )
                if ( Action.action_id.server_id = Server_id ) delete action from Ongoing_queue

Mark_yellow( Action )

        Mark_red( Action )
        Yellow.set = Yellow.set + Action


Mark_green( Action )

        Mark_red( Action )
        if ( Action not green )
                place action just on top of the last green action
                Green_lines[ Server_id ] = Action.action_id
                Apply_green( Action )
```

Figure 6.19: Code of the Marking Procedures.

Discussion

The replication algorithm eliminates the need for an end-to-end acknowledgment at servers level **without** compromising consistency. End-to-end acknowledgment is still needed after the membership of the connected servers is changed. Thus, the performance gain is substantial compared to all other techniques that use end-to-end acknowledgment at the servers level for each action. However, this unique merit does not come free (compared to [Kei94], for example). There exist two relatively rare scenarios where communication with every server of the last primary component is required before the next primary component can be formed:

1. Total crash: all of the servers in a primary component crash within a window of time so short, that the membership algorithm of the group communication could not be completed at any of them.
2. A group of servers tries to form a primary component and the network partitions just after each of them sent the CPC message, in such a way that **all** of them receive all the CPC messages, but not in the regular configuration. i.e. they **all** receive a regular configuration (Reg_conf event) while in the Un state (see Figure 6.17). In this case, the algorithm requires all the servers to remain vulnerable until at least one of them communicates with all the rest. This does not require direct connection since the eventual path propagation technique is used. A server which learns that this was the scenario, is no longer vulnerable, allowing for the next primary component to be formed (refer to step 4 of the Compute_knowledge procedure in Figure 6.12).

Clearly, any algorithm that overcomes processor crashes and recoveries, and avoids end-to-end acknowledgments per action, suffers from the first scenario. Regarding the second scenario, though rare, we hope to ease this requirement in a future development of this algorithm.

In this chapter we have focused on a service that complies with the correctness criteria defined in Chapter 2. For that, the Apply_red procedure is empty and the Apply_green procedure applies the action to the database. In the next section we prove that both correctness (safety and liveness) criteria hold. To prove the safety criterion we show that Apply_green procedure is invoked in the same order of actions at all the replication servers in the servers group. Other possible setups for the Apply_green and Apply_red procedures for various types of applications are possible, and are discussed in Chapter 7.

6.3 Proof of Correctness

In this section we prove that the replication protocol maintains the safety and liveness criteria defined in Chapter 2. We assume that the group communication layer maintains extended virtual synchrony.

Notations

- We say that an action *a* is performed (or reaches its final order) by server *s* when the Apply_green procedure is invoked for *a* at *s*. We say server *s* has action *a* when the Apply_red procedure is invoked for *a* at *s*.
- *S* - all of the members of the *servers group*.
- $a_{r,j}^{s,i}$ - action *a* is the *i*th action generated by server *s*, and the *j*th action performed (by Apply_green) by server *r*. Notations such as $a_{r,j}$ and $a^{s,i}$ are also possible where the generating/performing server is not important.
- The pair (*px,ax*) represents the *prim_index* and *attempt_index* of the *Prim_component* structure. We say that (*px,ax*) > (*px*',*ax*') iff either $px > px'$ or $px = px' \wedge ax > ax'$.
- $PC_s(px,ax)$ - server *s* installed or learned about primary component with *px* as primary_index and with *ax* as attempt index. Notations such as $PC_s(px)$, and $PC(px)$ are also possible when the missing parameters are not important.
- We say that server *s* is a member of *PC*(*px*) if *s* is in the servers set of *PC*(*px*) (therefore, *s* sent a CPC message that allowed the installation of *PC*(*px*)).

6.3.1 Safety

We prove that the following properties are invariants of the protocol. i.e. they are maintained throughout the execution of the protocol:

- **Global FIFO Order** - If server *r* performed an action *a* generated by server *s*, then *r* already performed every action that *s* generated prior to *a*.

  $a_{r,j}^{s,i} \Rightarrow$ for all $i' < i$ there exist $j' < j$ such that $a_{r,j'}^{s,i'}$.

- **Global Total Order** - If both servers *s* and *r* performed their *i*th actions then these actions are identical.

  $\exists a_{s,i}, a_{r,i} \Rightarrow a_{s,i} = a_{r,i}$.

Note that the global FIFO order and the global total order invariants imply that the global total order is also consistent with causal order.

We assume that all of the servers start with the following initial state:

Prim_component.prim_index=0, *Prim_component.attempt_index=0*, *Prim_component.servers = S,* empty *Actions_queue, Vulnerable.status*= Invalid, *Yellow.status*= Invalid.

Before proving the invariants, we will prove a few claims regarding the history of primary components in the system.

Claim 1. *If server r learns about* $PC_r(px,ax)$, *then there is a server s that installed* $PC_s(px,ax)$ such that $PC_r(px,ax) = PC_s(px,ax)$.

Proof: A server *r* knows about *PC*(*px, ax*) either when installing it or when learning about it. From the algorithm, the only place *r* learns about $PC_r(px,ax)$ is at Step 1 of the Compute_knowledge procedure (Figure 6.12). According to Step 1, there is a server *t* that sent a State message containing $PC_r(px,ax)$. Therefore, to start to chain, there must be a server *s* that installed $PC_s(px,ax)$ such that $PC_r(px,ax) = PC_s(px,ax)$. □

Claim 2: *The pair* (*px,ax*) *never decreases at any server s; Moreover, it increases each time server s sends a CPC message or installs a new primary component.*

Proof: Before a server installs a primary component, it sends a State message containing its last known primary component (Field *Prim_component* in the State message). Note that just before sending the State message, the server forces its data structure to disk, so that this information is not lost if the server crashes subsequently. In the Compute_knowledge procedure (Figure 6.12), the server sets *Prim_component* to the maximal (*px, ax*) that was sent in one of the State messages (including its own). Therefore, the local value of (*px,ax*) does not decrease.

Just before sending the CPC message, while the server is in the Exchange_actions State, it increments *Prim_component.attempt_index* and immediately forces its data structure to disk (Figure 6.10).

When installing, the server increments *Prim_component.prim_index* (Figure 6.15) and forces its data structure to disk. Since these three places are the only places where *Prim_component* may change, the claim holds. □

Claim 3: *If server s installs* $PC_s(px,ax)$, *then there exists server r that installed* $PC_r(px-1,ax')$.

Proof: According to the algorithm, if server *s* installs a primary component with *Prim_component.prim_index* = *px*, then there is a server *t* that sent a State message containing *Prim_component.prim_index* = *px*-1 for that installation. Therefore *t* either installed a primary component with *Prim_component.prim_index* = *px*-1 or learned about it. In any case, according to Claim 1, there exists a server *r* that installed such primary component. □

Claim 4: *If server s installed* $PC_s(px,ax)$ *and server r installed* $PC_r(px,ax')$, *then* $ax = ax'$ *and* $PC_s(px,ax) = PC_r(px,ax)$.

Proof: We prove this claim by induction on the primary component index *px*.

First we show that the claim holds for *px*=1:

Assume the contrary. Without loss of generality, suppose that $ax > ax'$. Remember that at initialization, the set of servers in *Prim_component* is *S*. Therefore, since *s* installs a primary component with (1,*ax*) there is a majority of *S* that participated in that attempt and sent a CPC message with (0,*ax*).

For the same reason, there is a majority of *S* that sent a CPC message with (0,*ax*'). Hence, there must be a server *t* that participated in both attempts and sent both messages. From Claim 2 and from the fact that $ax > ax'$, *t* sent the CPC message with (0,*ax*') before sending that with (0,*ax*).

From the algorithm, since *r* installed, there is a server that received all the CPC messages of the first attempt in the regular configuration. i.e. there is a server belonging to the first majority, that shifted from Construct to Reg_prim (Figure 6.14). The safe delivery property of extended virtual synchrony ensures that all the members of the first majority (including *t*) received all the CPC messages before the next regular configuration, or crashed. Therefore, according to the algorithm, for each server *u* belonging to the first majority, only the following cases are possible:

1. Server *u* receives all the CPC messages in the regular configuration and installs a primary component *PC*(1,*ax*') (see Figure 6.14).
2. Server *u* crashes before processing the next regular configuration and remains vulnerable.
3. Server *u* receives all the CPC messages, but some are delivered in the transitional configuration. In this case *u* is in the Un state. Two sub-cases are possible:

   3.1. Server *u* receives an action, and installs *PC*(1,*ax*') (see Figure 6.17).

   3.2. Server *u* receives the next regular configuration and remains vulnerable.

Since these are the only possible cases, every member of the first majority either installs *PC*(1,*ax*') or remains vulnerable. If server *t* installs PC(1,*ax*'), then Claim 2 contradicts the fact that it later sent a CPC message with (0,*ax*). If server *t* remains vulnerable, then according to the algorithm, it must invalidate its vulnerability before sending another CPC message. This can happen in the only following ways:

1. Server *t* learns about a higher primary component. Again, Claim 2 contradicts the fact that it later sent a CPC message with (0,*ax*).
2. Server *t* learns that all the servers from the first majority did not install and are vulnerable to the same server set, contradicting the fact that server *r* installed $PC_r(px, ax')$.
3. Server *t* learns that another server from the set is not vulnerable with the same (0,*ax*'). The only servers that are not vulnerable in this set, are the servers that installed. Hence server *t* learns about the installation of a higher primary component before sending its second CPC message, which contradicts Claim 2.

Therefore, no such server *t* exists, proving the base of the induction.

The induction step assumes that the claim holds for *px* and shows that it holds for *px*+1.

The proof is exactly the same proof as for *px*=1, where 0 is replaced by *px*, 1 is replaced by *px*+1, and *S* is replaced by the set of servers in *PC*(*px,ax*). □

Claim 5: *If server s installed or learned about* $PC_s(px)$ and *server r installed or learned about* $PC_r(px)$, *then* $PC_s(px) = PC_r(px)$.

Proof: Follows directly from Claim 1 and Claim 4. □

From here on, we will note *PC*(*px*) the primary component that was installed with *prim_index px*. Claim 5 proves the uniqueness of *PC*(*px*) for all *px*.

Claim 6: *If a primary component PC*(*px*+1) *is installed, then a primary component PC*(*px*) *was already installed, and there exists a server s such that s is a member of both sets of PC*(*px*) and *PC*(*px*+1).

Proof: If a primary component *PC*(*px*+1) is installed then there exists a server that installed it. According to Claim 3, there is a server that installed *PC*(*px*). According to Claim 5, all the servers that installed or learned about a primary component with index *px*, installed or learned about the same *PC*(*px*). According to the algorithm, a majority from *PC*(*px*) is needed to propose *PC*(*px*+1) in order to install *PC*(*px+1*), therefore there is a server (actually, a majority of servers) that are members of both sets of *PC*(*px*) and *PC*(*px*+1). □

We are now ready to prove the invariants. There are three places where the Mark_green procedure is called. We label them with the following labels that appear also in the protocol pseudo-code:

1. *OR-1* - the action was sent and delivered at a primary component.
   - ⇒ *OR-1.1* - the action was delivered at the regular configuration of a primary component (see Figure 6.7).
   - ⇒ OR-1.2 - the action was delivered at the transitional configuration of the primary component, for each member of the last primary component that participates in the quorum that installs the next primary component (see Figure 6.15).
2. *OR-2* - The action was delivered at a non-primary component and was ordered with the installation of a new primary component (see Figure 6.15).
3. *OR-3* - The action was ordered when this server learned about this order from another server that already ordered it (see Figure 6.11).

Claim 7: *If server s orders an action according to OR-3 then, there exists server r that order this action at the same order according to OR-1 or OR-2.*

Proof: At initialization, no actions are ordered at any of the servers in *S*, since *Actions_queue* is empty. *OR-1*, *OR-2* and *OR-3* are the only possible places where actions are ordered by the algorithm.. If server *s* orders action *a* according to *OR-3* then there was a server that multicast the action *a* and its order at the Exchange_actions state (see Figure 6.9 and Figure 6.11). To start this chain, there must be a server that ordered action *a* in a different way, and *OR-1* and *OR-2* are the only possibilities. □

Claim 8: *Assume that: servers s and r install PC(px), they have the same set of actions, ordered in the same order, and marked in the same color, in their Action_queue when sending the CPC message for PC(px). Then, for every action a that both r and s ordered in PC(px) according to OR-1.1, they ordered it at the same order.*

Proof: Under the assumption, and according to the algorithm *s* and *r* have identical *Actions_queue, Green_lines* and invalid *Yellow* when they complete the Install procedure.

Since *a* was ordered both at *r* and *s* according to *OR-1.1* it was delivered both to *r* and *s* in the regular configuration *c* within which *PC*(*px*) existed. According to the agreed delivery property of extended virtual synchrony, the same set of actions at the same order is delivered up to and including action *a* to both *r* and *s*. According to the algorithm, each delivered action in the Reg_prim state is marked green immediately when it is delivered. Therefore, both *r* and *s* ordered the action *a*, and all previous actions, at the same order. □

Claim 9: *Assume that: servers s and r are members of PC(px), they have the same set of actions, in the same order, and marked in the same color in their Actions_queue, when sending the CPC message for PC(px), and s is a member of PC(px+1). Then, for every action a that r ordered in PC(px) according to OR-1.1, s either ordered a or has a in its Yellow at the same order before sending the CPC message for PC(px+1). Moreover, if s installs PC(px+1) than s ordered a at the same order as r.*

Proof: Under the assumption, and according to the algorithm *s* and *r* have identical *Actions_queue, Green_lines* and invalid *Yellow* when they complete the Install procedure.

Since *a* was ordered at *r* according to *OR-1.1* it was delivered to *r* in the regular configuration *c* within which *PC*(*px*) existed. According to the safe delivery property of extended virtual synchrony, *a* was delivered in *c* or in *trans*(*c*) to every server *t* in *PC*(*px*) unless it crashes. According to the agreed delivery property, *a* and all prior messages in configuration *com*(*c*) were delivered to *t* in the same order.

Therefore, only three cases are possible for any server *t* in *PC*(*px*):

1. Action *a*, and all previous actions delivered to *r* in *c*, are delivered to *t* in the regular configuration *c*. According to the algorithm, *t* marks *a* and all previous actions as green at the same order (see Figure 6.7) according to *OR-1.1*.

2. Action *a*, and all previous actions delivered to *r* in *c*, are delivered to *t* in the regular configuration *c* or transitional configuration *trans*(*c*) in the same order, and *a* is delivered in the transitional configuration *trans*(*c*), and the next regular configuration is delivered and processed at *t*. In this case, according to the algorithm, *a* and all prior actions that are not yet ordered will be included in the *Yellow* at the same order, and *Yellow* will be valid.
3. Server *t* crashed before action *a* was processed at *t* and before the next regular configuration was processed at *t*. According to the algorithm, server *t* remains vulnerable after it recovers.

Consider server *s*. If Case-1 exists for *s*, then *s* already ordered *a* at the same order as *r* before sending the CPC message for *PC*(*px*+1).

If Case-2 exists for *s*, then action *a* and all previous actions in *Yellow* are not extracted from the *Yellow* of *s* unless *s* learns about the order of it before sending the CPC message for *PC*(*px*+1). In this case, according to Claim 7, there is a server *u* that already ordered this message according to *OR-1* or *OR-2*. The only possibility is that *u* is a Case-1 server so *a* and all previous actions were ordered at *u* at the same order as *r*. According to the algorithm, *s* ordered *a* and all previous actions at the same order as *r* at the Exchange_actions state (see Figure 6.11).

If Case-3 exists for *s* then, according to the algorithm (see Figure 6.13) *s* has to invalidate its vulnerability before it is able to send a CPC message to install *PC*(*px*+1). There are only two possibilities for *s* to invalidate its vulnerability. The first possibility is to learn that another server that belongs to *PC*(*px*) is unvulnerable. The only unvulnerable servers are Case-1 and Case-2 servers which are more updated than *s*. If *s* learns about them and their order (see Figure 6.9 and Figure 6.11), then the claim holds. The second possibility occurs if *s* learns that all of the servers that belong to *PC*(*px*) are vulnerable. In this case, according to the algorithm, they all crashed before processing the next regular configuration. Since there exists at least one Case-1 server (*r*), the most updated server in PC(*px*) is a Case-1 server. Therefore, according to the algorithm (see Figure 6.9 and Figure 6.11), when *s* learns that *r* is also vulnerable, *s* also learns the order of *a* and all the previous actions at *r*.

Lastly, if *s* installs *PC*(x+1), it mark all the yellow actions as green. □

Claim 10: *If server s is a member of PC*(*px*) *then:*

(i) s has marked as green or yellow at the same order, every action that any server marked as green according to OR-1 or OR-2 in PC(*px*-1).

(ii) s has marked as green at the same order, every action that any server marked as green according to OR-1 or OR-2 in PC(*px*') *where px*'<*px*-1.

(iii) if r is another member of PC(*px*), *then r and s have the same set of actions, ordered in the same order, and marked in the same color, in their Actions_queue, when sending the CPC message for PC*(*px*).

Proof: We prove this claim by induction on the primary component index *px*.

First we show that the claim holds for px=1.

At initialization, no actions are ordered at any of the servers in S and *Yellow* is empty, proving (*i*) and (*ii*) for px=1. According to the algorithm, all the members of $PC(1)$ exchange actions so that they have identical set of actions in their *Actions_queue* before sending the CPC message, and all actions are red. Therefore, the claim holds for px=1.

The induction step assumes that the claim holds for all primary components up to and including px and shows that the claim holds for px+1.

According to the algorithm and to the basic delivery and delivery of configuration change properties of extended virtual synchrony, only members of $PC(px)$ can order actions according to OR-1 or OR-2 in $PC(px')$ for any px'.

According to Claim 6, there is one server t that is a member of both $PC(px)$ and $PC(px+1)$. Let s be any member of $PC(px+1)$. Only the following two cases are possible:

1. Server s was a member of $PC(px)$. According to the induction assumption (*iii*) s and t had the same set of actions in the same order, and marked in the same colors when sending the CPC message for $PC(px)$. According to Claim 9, and since s and t are members of $PC(px)$, they both had marked as green or yellow, at the same order, all actions that **any** server marked as green according to *OR-1* or *OR-2* in $PC(px)$. Thus, (*i*) is proved. Moreover, according to the induction assumption (*ii*) and Claim 9, and the fact that there is a server that installed $PC(px)$, they both marked as green, at the same order, all actions that **any** server marked as green according to *OR-1* or *OR-2* in $PC(px')$ for any $px' \prec px$. Thus, (*ii*) is proved. Finally, According to the algorithm, and since they are both members of $PC(px+1)$, they both sent a CPC message for $PC(px+1)$, which required them to go through a retransmission phase in the Exchange_actions state (see Figure 6.11). Therefore, they have the same actions, ordered at the same order and marked at the same colors before sending the CPC message for $PC(px+1)$. Thus, (*iii*) is proved.

2. Server s was **not** a member of $PC(px)$. According to the induction assumption (*ii*) t has marked as green at the same order all actions that s marked green according to *OR-1* and *OR-2*, proving (*i*) and (*ii*). According to Claim 7, any action ordered by s according to *OR-3* was already ordered at the same order by another server according to *OR-1* or *OR-2*. According to the induction assumption (*i*) and (*ii*), t has marked as green or yellow at the same order as any other server that ordered it as green. Hence, t's order can not contradict s's order. Since they exchange actions before sending the CPC they will have the same set of red, yellow and green before sending their CPC messages. Thus, (*iii*) is also proved. □

Theorem 11: Global Total Order: *If both servers s and r performed their ith actions then these actions are identical.*

$$\exists a_{s,i}, a_{r,i} \Rightarrow a_{s,i} = a_{r,i}.$$

Proof: From Claim 10 and Claim 8, all the servers that order an action according to *OR-1* or *OR-2* do so in the same order. From Claim 7, if a server orders an action according to *OR-3* there already exists a server that ordered that action according to *OR-1* or *OR-2* at the same order. Therefore, since *OR-1*, *OR-2*, and *OR-3* are the only possibilities to order action, if two servers ordered an action *a* they ordered *a* at the same order. □

Claim 12: *If r has* $a^{s,i}$ such that $a^{s,i}$ *was generated by s at connfiguration c and was delivered to r at com(c) then, r already has* $a^{s,j}$ *for any* $j<i$.

Proof: According to extended virtual synchrony, both *s* and *r* are members of *c*.

According to the algorithm, when a regular configuration is delivered, then the servers exchange actions that are missed by any of them (see Figure 6.9 and Figure 6.11). If server *s* generated $a^{s,j}$ before this retransmission then, if *r* does not have $a^{s,j}$, it will be retransmitted. If server *s* generated both $a^{s,j}$ and $a^{s,i}$ after the retransmission, then according to the algorithm, $a^{s,j}$ was generated first. According to the causal delivery property of extended virtual synchrony, if $a^{s,i}$ is delivered to *r* in *com(c)* then $a^{s,j}$ is delivered to *r* before $a^{s,i}$. □

Theorem 13: Global FIFO Order: *If server r performed an action a generated by server s, then r already performed every action that s generated prior to a.*

$$a^{s,i}_{r,j} \Rightarrow \text{for all } i' < i \text{ there exist } j' < j \text{ such that } a^{s,i'}_{r,j'}.$$

Proof: According to the algorithm, *s* creates its own actions according to a FIFO order. Moreover, *s* never looses its own actions even if it crashes (see Figure 6.6 and Figure 6.7 and Figure 6.18).

Assume the contrary. Without loss of generality, assume that *t* is the first server that orders the *i*th action of some server *s*, $a^{s,i}$, such that the *j*th action of *s*, $a^{s,j}$, is not ordered at *t* for some $j<i$. Therefore, according to Theorem 11, any server that orders $a^{s,i}$, orders it before ordering $a^{s,j}$.

Since *t* is the first server to order $a^{s,i}$, only three cases are possible for *t*:

1. Server *t* orders $a^{s,i}$ according to *OR-1.1*. In this case, according to the algorithm, $a^{s,i}$ is delivered in a primary component *PC*(*px*) such that *t* is a member of *PC*(*px*). According to properties 2.1 and 1.3 of extended virtual synchrony, *s* is also a member of the regular configuration within which *PC*(*px*) is installed. Therefore, according to the algorithm, *s* is also a member of *PC*(*px*). Therefore, according to Claim 12, $a^{s,i}$ was delivered (and therefore, ordered) first. i.e. this case is not possible.

2. Server *t* orders $a^{s,i}$ according to *OR-1.2*. In this case, according to the algorithm, there was a *u* server at which $a^{s,j}$ was delivered in a transitional configuration of some primary component *PC*(*px*). According to base delivery property of extended virtual synchrony, and to the algorithm, *s* is a member of *PC*(*px*). If $a^{s,j}$ was generated before the installation of *PC*(*px*), then *u* ordered it at the installation of *PC*(*px*), and before ordering $a^{s,i}$. If $a^{s,j}$ was generated after the installation of PC(px) then both $a^{s,i}$ and $a^{s,j}$ where delivered in the same configuration. According to the causal delivery property of extended virtual synchrony, $a^{s,i}$ was delivered (and therefore, ordered) first. i.e. this case is not possible.

3. Server *t* orders $a^{s,i}$ according to *OR-2*. In this case, according to the algorithm, *t* orders all its unordered actions according to their *Action_id* (see Figure). Since $j<i$, and both $a^{s,i}$ and $a^{s,j}$ are generated by the same server, if *t* had $a^{s,j}$ then it would have ordered it before $a^{s,i}$. Therefore, *t* does not have $a^{s,j}$.

Therefore, since only Case-3 might be possible, there has to be a server that receives $a^{s,i}$ **before** it received $a^{s,j}$ and **before** $a^{s,i}$ is ordered by any server. Assume that *r* is the first server to receive $a^{s,i}$ before it received $a^{s,j}$.

According to Claim 12, Server *r* could not have $a^{s,i}$ as a new action generated by *s* without having all prior actions generated by *s*. Therefore, according to the algorithm, the only option left for *r* is to have $a^{s,i}$ as a result of a retransmission. Since $a^{s,i}$ is not yet ordered, and each server that has $a^{s,i}$ has also $a^{s,j}$, and since retransmission for unordered messages is done in FIFO order, $a^{s,j}$ is retransmitted first. By causal delivery property of extended virtual synchrony, $a^{s,j}$ is delivered to *r* before $a^{s,i}$, leading to a contradiction. □

6.3.2 Liveness

To prove liveness of the protocol, we assume two properties regarding behavior of the group communication layer.

1. If there exists a set of processes containing *s* and *r*, and a time, from which on that set does not face any communication or process failure, then the group communication eventually delivers a configuration change containing *s* and *r*. Moreover, we assume that if no message is lost within this set, the group communication will not deliver another configuration change to *s* or *r*.
2. If a message is deliverable then the group communication layer eventually delivers it.

We would like to note that Transis, for example, does behave according to these assumptions.

Theorem 14: Liveness: *If server s orders action a and there exists a set of servers containing s and r, and a time, from which on, that set does not face any communication or process failures, then server r eventually orders action a.*

$$\Diamond(\exists a_{s,i} \wedge \Box\, stable\_system(s,r)) \Rightarrow \Diamond\exists a_{r,i}\,.$$

Proof: Since there exists a set of servers containing *s* and *r*, and a time from which on that set does not face any communication or process failures then, according to Assumption 1 on the group communication, there is a time at which the group communication delivers a configuration change *c* containing *s* and *r* to both *s* and *r*, and from this time on does not deliver any other configuration change to *s* and *r*.

If *a* is ordered at *r* at the time *c* is delivered to *r* then, according to Theorem 11, *r* orders *a* at the same order *s* ordered *a*.

Assume *a* is not ordered at *r* at the time *c* is delivered to *r*. There are only two cases:

1. Server *s* ordered action *a* before the delivery of *c*. In this case, after the *c* is delivered, according to the algorithm, *s* and *r* send State messages, and exchange actions and knowledge (see Figure 6.11). Since *s* already ordered action *a* the most updated server already ordered *a*. Moreover, according to Theorem 11, *a* was ordered at the most updated server at the same order as in *s*. According to Assumption 2 on the group communication, all these messages are delivered to *r*. Therefore, *a* and its order are eventually delivered to *r*. According to the algorithm, *r* orders *a* (*OR-3*). According to Theorem 11, *r* orders *a* at the same order *s* ordered *a*.
2. Server *s* ordered action *a* after the delivery of *c*. Therefore, since no other configuration are delivered after *c*, a primary component *PC*(*px*) is created within *c*, with s and r as members. Since there are no communication or server failures, and by Assumption 2 on the group communication layer, both *s* and *r* eventually install *PC*(*px*). Since *s* ordered action *a* at that primary component, only three sub-cases are possible:
   2.1. Server *s* ordered *a* according to *OR-1.2*. Therefore, *a* was a yellow action at *s* which was ordered when installing *PC*(*px*) (see Figure 6.15). According to Assumption 2 on the group communication, eventually, *r* has the same set of actions after the retransmissions and gets all the CPC messages. Therefore, *r* eventually installs *PC*(*px*). According to the algorithm, when installing, *r* orders its yellow actions, including action *a*. According to Theorem 11, *r* orders *a* at the same order *s* ordered *a*.
   2.2. Server *s* ordered *a* according to *OR-2*. Therefore, *a* was a red action at *s* which was ordered when installing *PC*(*px*) (see Figure 6.15). According to Assumption 2 on the group communication, *r* eventually installs *PC*(*px*), and *r* also has the same set of actions after the retransmissions. According to the algorithm, *r* also orders the red action *a* according to *OR-2*. According to Theorem 11, *r* orders *a* at the same order *s* ordered *a*.

2.3. Server *s* ordered *a* according to *OR-1.1*. Therefore *a* is delivered to *s* at configuration *c* (see Figure 6.7). Since there are no communication or server failures, *a* is also deliverable at the group communication layer of *r*. According to assumption 2 on the group communication layer, *a* is eventually delivered to *r*. According to the algorithm *r* immediately orders *a*, and according to Theorem 11, it orders *a* at the same order *s* ordered *a*. □

Chapter 7

7. Customizing Services for Applications

This chapter shows how to use the replication server in order to tailor optimized services for different types of applications.

Many applications require that the replicated database will behave as if there is only one copy of it (as far as the application can tell). We say that such applications require *strict consistency* semantics. In the previous chapter, we saw that the green order preserves our safety criterion, assuring one-copy serializability for databases that comply with our service model.

In the primary component, actions are marked green and are applied to the database immediately. Applications that require strict consistency can assume the strong property of one-copy serializability. They get good response while in the primary component, **but** have to pay the cost of being blocked while not in the primary component.

In the real world, however, where incomplete knowledge is inevitable, many applications would choose to have an immediate reply, rather than incur a long latency to obtain a complete and consistent reply. Therefore, we provide additional services for clients in a non-primary component.

A *weak consistency query*, or simply a weak query, results in an immediate reply derived from the (consistent) database state reflected by all the green actions. Although the weak query yields results derived from a state that was consistent, it may now be obsolete. In particular, a client may initiate updates and then issue a weak query to find that the updates are not reflected in the result of the query. Therefore, this service is not consistent.

A *dirty query* results in an immediate reply, derived from the (inconsistent) database state reflected by the green **and the red** actions. This service is not consistent because the result of the query reflects actions that are not yet committed. The semantics of a dirty query resembles that of a dirty read in a non-replicated database. As defined in [GR93], a dirty read may reflect updates performed by transactions that are not yet committed. Dirty query is useful when an immediate, "to-the-best-of-your-knowledge" reply is desired.

It is important to note that while in the primary component, the results of the weak query and the dirty query are identical to those of the strictly consistent query.

Some applications are indifferent to the order in which actions are applied to the database. If the update semantics is restricted to commutative updates, for example, we can optimize the service.

The reminder of this chapter tailors the relevant services for each of the above semantics. The services are defined by customizing the Apply_red and Apply_green procedures of the replication server.

7.1 Strict Consistency

Strict consistency is preserved by applying the action to the database only when it is marked green. Therefore, the Apply_red procedure is empty.

The Apply_green procedure applies the action to the database. The action's update is applied to the database at each of the replication servers. However, only the replication server that received the original request from the client queries the database and sends the result back to the client. Figure 7.1 presents the pseudo-code of the Apply_green and Apply_red procedures when strict consistency is maintained.

```
Apply_red( Action )                    Apply_green( Action )

  exit                                   apply Action.update to Database
                                         if ( Server_id = Action.action_id.server_id)
                                                 apply Action.query to Database
                                                 return reply to Action.client
```

Figure 7.1: Maintaining Strict Consistency.

Since the action is applied only after its global order is determined, when the server is not in a primary component, the action (as well as the query) is blocked. When the query is finally replied, the result is consistent.

Computing the monthly interest for bank accounts is a good example of the need for such service. When querying the balance over which the interest is computed, it is important that the result will reflect all the actions that are ordered before the query and nothing else. Since the interest update is not commutative with withdrawals and deposits, the application has to wait for a consistent balance.

Optimization for Queries

When the action contains only a query (the update part is empty), the replication server need not generate a message. Instead, the server can apply this action to the database as soon as all previous actions generated by this server are already applied. This method can be further optimized to apply the action as soon as all previous actions requested by the creating client are already applied (assuming that clients are not creating causal dependencies outside our system). Similar optimization appears in [KTV93].

Active Actions

It may be useful for many applications to have the ability to process an action by executing a procedure specified by the action. This option enhances the service semantics of updates, as was defined in Chapter 2. This extension does not affect the safety and liveness criteria defined there, provided that the invoked procedure is deterministic and depends solely on the current database state. The key is that the procedure is invoked only at the time the action is ordered, rather than invoked before the creation of the update.

Interactive Actions

Our model best fits one-operation transactions. However, several applications need to provide the client with the ability to apply interactive complex transactions. For example, within one transaction, a client program may need to read a few values, then the user will make a decision, and then the update will be applied, and after that the transaction will try to commit.

This behavior cannot be implemented in our approach using one action. However, it can be mimicked with the help of two actions. In the first action, the necessary information is read. A second distinct action is actually an active action (as in the above sub-section). The active action invokes a procedure which first checks whether the values of the data read by the first action are still valid (identical). If so, the update will be applied to the database. If not, the update will not be applied (as if the action is aborted in a traditional database). Note that if one server "aborts" then all of the servers "abort" that (trans)action, since they apply an identical deterministic rule to an identical state of the database.

7.2 Weak Consistency Query

In the primary component, actions are marked green and are applied to the database immediately. However, in order to be consistent, actions have to be blocked while not in the primary component. In principle, if no updates are generated while in a non-primary component, clients can freely query the database and get consistent replies. However, as we saw in previous chapters, a client cannot tell a priori whether its requested action will be delivered in a primary component even if the client belong to a primary component at the time the request is made. Moreover, since there is no common knowledge, even the replication server cannot tell whether the generated message will be delivered as safe in the regular configuration at the time it is multicast. Hence, updates cannot be confined to a primary component unless no updates are allowed, or unless the selection method of a primary component is monarchy.

For some applications, it might be beneficial to get an immediate reply, based on a consistent (yet possibly old) state of the database. To choose this option, the client requests a weak consistent query. If this query is applied in the primary component, it

results in a strict consistent reply. Figure 7.1 presents the pseudo-code of the Apply_green and Apply_red procedures designed to maintain weak consistency semantics.

```
Apply_red( Action )

  if ( in primary now ) exit
  if ( Server_id = Action.action_id.server_id and
       Action.update is empty )
          apply Action.query to Database
          return reply to Action.client
          mark action as replied
```

```
Apply_green( Action )

  apply Action.update to Database
  if ( Server_id = Action.action_id.server_id )
       if ( Action not replied )
            apply Action.query to Database
            return reply to Action.client
```

Figure 7.2: Maintaining Weak Consistency.

As mentioned earlier, a weak consistent query potentially violates one-copy serializability. However, it is important to note that since updates are applied in the same order at all of the replication servers, the databases converge to the same state after the same set of updates is applied. Later, we show that weak consistent queries can coexist with strict consistent actions, allowing users to choose if they want a strictly consistent reply and are willing to be blocked (if not in the primary component), or rather have an immediate reply based on some old value (if not in the primary component).

In principle, the replication server can immediately apply weak queries without even generating a message and still keep the described weak consistency semantics. We choose not to do that in order to allow weak consistent queries to return a strict consistent reply while in a primary component, and for the sake of simplicity.

7.3 Dirty Query

Many applications would rather get an immediate reply based on the latest information known. In the primary component, the latest information is consistent. However, in a non-primary component, red actions must be taken into account in order to provide the latest, though not consistent, information. Dirty query is useful when an immediate, “to-the-best-of-your-knowledge” reply is desired.

Computing the monthly interest for bank accounts is a good example of the need for such a service. When querying the balance over which the interest is computed, it is important that the result will reflect all the actions that are ordered before the query and nothing else. Since the interest update is not commutative with withdrawals and deposits, the application has to wait for a consistent balance.

When a cash withdrawal is made, using an automatic teller machine connected to a server which is not in a primary component (i.e. it is currently disconnected from the server managing the account), this server cannot return a consistent reply regarding the new account balance. Since the user is not likely to wait for a consistent balance, the

server may return both the weak consistency balance, valid for the previous business day, and the dirty balance, computed based on the weak consistency balance and later withdrawals known to this server (at least, in Israel, most banks give these two balances on the withdrawal receipt).

In order to provide a dirty read service, while maintaining convergence of the database to a consistent state, a *dirty version* of the database reflecting unordered red updates is kept while in a non-primary component. In most cases, the preferred way to manage the dirty version is to maintain a *Delta* database containing the records affected by the red updates. A dirty query may be replied by scanning the Delta database and the database. This way, the results of a dirty query are based on a database state reflected by all the known updates. Of course, maintaining the Delta database is an application dependent task. The replication server simply tells the application which updates and which queries to apply to which version of the database, and instructs the application to delete the dirty version of the database when a primary component is installed and the database state converges to a consistent state.

Figure 7.3 presents a combined scheme that allows the user to select the desired query type (consistent, weak, or dirty) to be invoked when applied in a non-primary component.

```
Apply_red( Action )

  if ( in primary now ) exit
  apply action.update to Delta
  if ( Server_id = Action.action_id.server_id )
      if( weak required )
            apply Action.query to Database
            return reply to Action.client
            mark action as replied
      if( dirty required )
            apply Action.query to Delta+Database
            return reply to Action.client
            mark action as replyed
```

```
Apply_green( Action )

  if ( Delta exist ) Delete Delta
  apply Action.update to Database
  if ( Server_id = Action.action_id.server_id )
        if ( Action not replied )
              apply Action.query to Database
              return reply to Action.client
```

Figure 7.3: A Combined Scheme.

7.4 Timestamps and Commutative Updates

Many times, a restricted semantics of the update model can be exploited in order to optimize the service latency. This section focuses on two such restrictions: The *timestamp* update semantics, and the *commutative* update semantics.

In timestamp update semantics, each record in the database maintains a timestamp. Each update overwrites a previous version that has an older timestamp. Alternatively, it can be augmented to a sorted list of record versions representing the history of that record. An example of this semantics is location tracking of taxis. Suppose that each taxi

tracks its location using a Global Position System and periodically broadcasts its identifier, location, and current time. Several servers, perhaps located at different sites, receive updates broadcast by taxis local to them. Each server builds its view of the taxis' position over time. Obviously, two servers that receive the same set of updates have the same database state. Using this semantics, each action is applied immediately when received by the replication service.

In a commutative update semantics, the order by which actions are applied is not important, as far as the database state is concerned. Here, again, a similar approach can be taken. An example of this semantics is an inventory management, where the operations are restricted to inserting items, extracting items, and querying the amount in the inventory.

Figure 7.4 presents a scheme, optimized for the timestamp semantics and for the commutative update semantics.

```
Apply_red( Action )                                Apply_green( Action )

  apply Action.update to Database                    exit
  if ( Server_id = Action.action_id.server_id )
          apply Action.query to Database
          return reply to Action.client
```

Figure 7.4: An Optimized Scheme for Timestamps and Commutative Updates.

Note that with the timestamp semantic or the commutative update semantic, one copy serializability is not maintained in case partitions occur. However, after the network is repaired and the partitioned components merge, the database states converge.

The timestamp and the commutative update semantics are simplified versions of the Read-Independent Timestamped Updates (RITU) and Commutative Updates (COMMU) semantics defined in the seminal work of [PL91].

7.5 Discussion

In our opinion, whenever an application can be restricted to the timestamps or commutative updates, the above solution should be followed. Even when this restriction does not fully comply with system requirements, it is advisable to weigh the problems arising from addressing the semantics differences, against the problems (and cost) of the more general solution. This model converts the replica control problem to the easier problem of guaranteeing the delivery of all updates to all of the replicas. The eventual path dissemination technique (see Chapter 6) presents an elegant solution for that problem.

Chapter 8

8. Conclusions

Replication is valuable for improving performance and availability of information systems. Client-server systems with replicated data may be able to provide better performance by sharing the queries' load between multiple servers. Replication also improves availability of information when servers may crash or when the network may partition.

This thesis presented a highly efficient architecture for replication over a partitioned network. The architecture is structured into two layers: a replication layer and a group communication layer. The architecture overcomes network partitions and re-merges, process crashes and recoveries, and message omissions.

We presented Transis, a group communication layer that utilizes the available non-reliable hardware multicast for efficient dissemination of messages to a group of processes. The Ring reliable multicast protocol described here, is one of the two protocols Transis uses to provide reliable multicast and membership services. The protocol's exceptional performance, over a network of sixteen Pentium machines connected by Ethernet, is demonstrated. Transis, developed at the Hebrew University of Jerusalem, has been operational for almost three years now. It is used by students in the distributed systems course, and by the members of the High Availability Lab. Several projects were implemented on top of Transis, among them a highly available mail system, a distributed system management tool, and several graphical demonstration programs. The Ring protocol was developed in the Totem project at the University of California, Santa Barbara.

We formulated the extended virtual synchrony semantics that defines the group communication transport services. Extended virtual synchrony supports continued operation in all components of a partitioned network. The significance of extended virtual synchrony is that during network partitioning and re-merging, and during process failure and recovery, it maintains a consistent relationship between the delivery of messages and the delivery of configuration change notifications across all processes in the system. Extended virtual synchrony provides well-defined self-delivery and failure atomicity, as well as causal, agreed and safe delivery properties. Both Transis and Totem provide the extended virtual synchrony semantics.

We constructed the replication server that provides long-term replication services within a fixed set of servers. Each of the replication servers maintains a private copy of the database. Actions requested by the application are globally ordered in a symmetric way by the replication servers, and are then applied to the database.

To efficiently propagate actions and knowledge between servers, we designed the *propagation by eventual path* technique. This technique optimizes the retransmission of actions and knowledge acquired within different components of the network, according to the configuration changes in the network membership. When a merge occurs in the network, servers from different components exchange information, where each action that is known to any of the servers and is missed by another server, is retransmitted exactly once.

We have constructed a global action ordering algorithm of the replication server. The novelty of this algorithm is the elimination of the need for end-to-end acknowledgments and for synchronous disk writes on a per-action basis. This elimination was made possible by utilizing the safe delivery service defined by the extended virtual synchrony semantics. Safe delivery provides stronger guarantees compared to the total order delivery, usually provided by group communication layers. End-to-end acknowledgment and synchronous disk writes are still needed, but only on a change in the membership of the connected servers. As a consequence, the replication server in a primary component applies actions to the database immediately on their delivery by the group communication layer, without the need to wait for other servers. This is done **without** compromising consistency.

Lastly, we showed how to use the replication server in order to tailor optimized services for different types of applications: Applications requiring strict consistency; applications requiring an immediate, though not necessarily consistent, reply for queries; and applications with a weaker update semantics (e.g. commutative updates). We also showed how the architecture may support active actions and interactive transactions.

High performance of the architecture is achieved because:

- Hardware multicast is used where possible.
- Synchronous disk writes are almost eliminated, without compromising consistency.
- End-to-end acknowledgments are not needed on a regular basis. They are used only after membership change events such as processor crashes and recoveries, and network partitions and merges.

Bibliography

[Aga94] D. A. Agarwal. Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks. Ph.D. thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, 1994.

[AAD93] O. Amir, Y. Amir and D. Dolev. A Highly Available Application in the Transis Environment. In *Proceedings of the Workshop on Hardware and Software Architectures for Fault Tolerance*, pages 125-139, Lecture Notes in Computer Science 774, June 1993.

[ADKM92a] Y. Amir, D. Dolev, S. Kramer and D. Malki. Transis: A Communication Sub-system for High Availability. In *Proceedings of the 22nd Annual International Symposium on Fault Tolerant Computing*, pages 76-84, July 1992.

[ADKM92b] Y. Amir, D. Dolev, S. Kramer and D. Malki. Membership Algorithms for Multicast Communication Groups. In *Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 292-312, Lecture Notes in Computer Science 647, November 1992.

[ADMM94] Y. Amir, D. Dolev, P. M. Melliar-Smith and L. E. Moser. Robust and Efficient Replication Using Group Communication. Technical Report CS94-20, Institute of Computer Science, The Hebrew University of Jerusalem, 1994.

[AMMAC93] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal and P. Ciarfella. Fast Message Ordering and Membership Using a Logical Token-Passing Ring. In *Proceedings of the IEEE 13th International Conference on Distributed Computing Systems*, pages 551-560, May 1993.

[AMMAC95] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal and P. Ciarfella. The Totem Single-Ring Ordering and Membership Protocol. In *ACM Transactions on Computer Systems,* to appear.

[BCJM+90] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck and M. Wood. The ISIS System Manual, Department of Computer Science, Cornell University, September 1990.

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

[BJ87] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the ACM Symposium on Operating Systems Principles,* pages 123-138, November 1987.

[BvR94] K. Birman and R. van Renesse. Reliable Distributed Computing with the ISIS Toolkit, Los Alamitos, CA., IEEE Computer Society Press, 1994.

[CM84] J. M. Chang and N. F. Maxemchuk. Reliable Broadcast Protocols, *ACM Transactions on Computer Systems*, 2(3):251-273, August 1984.

[CS93] D. R. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 44-57, December 1993.

[CS95] F. Cristian and F. Schmuck. Agreeing on Processor Group Membership in Asynchronous Distributed Systems. Technical Report CSE95-428, University of California at San Diego.

[CZ85] D. Cheriton and V. Zwaenepoel. Distributed Process Groups in the V-Kernel, *ACM Transactions on Computer Systems*, 3(2):77-107, 1985.

[Dee89] S. E. Deering. Host Extensions for IP Multicasting. RFC 1112, SRI Network Information Center, August 1989.

[EGLT76] K. Eswaran, J. Gray, R. Lorie and I. Taiger. The Notions of Consistency and Predicate Locks in a Database System. *Communication of the ACM*, 19(11), pages 624-633, 1976.

[ESC85] A. El Abbadi, D. Skeen and F. Cristian. An Efficient Fault-Tolerant Algorithm for Replicated Data Management. In *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 215-229, March 1985.

[ET86] A. El Abbadi and S. Toueg. Availability in Partitioned Replicated Databases. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 240-251, March 1986.

[FLP85] M. Fischer, N. Lynch and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32, pages 374-382, April 1985.

[Gif79] D. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the ACM Symposium on Operating Systems Principles,* pages 150-159, December 1979.

[Gol92] R. A. Golding. Weak Consistency Group Communication and Membership. Ph.D. thesis, Computer and Information Sciences Board, University of California at Santa Cruz, 1992.

[Gra78] J. Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course*, pages 393-481, Lecture Notes in Computer Science 60, Springer-Verlag, 1978.

[JM87] S. Jajodia and D. Mutchler. Dynamic Voting. In *Proceedings of the ACM SIGMOD International Conference of Management of Data*. pages 227-238, 1987.

[JM90] S. Jajodia and D. Mutchler. Dynamic Voting Algorithms for Maintaining the Consistency of Replicated Database. *ACM Transactions on Database Systems*, 15(2):230-280, June 1990.

[KD95] I. Keidar and D. Dolev. Increasing the Resilience of Atomic Commit, at No Additional Cost. *ACM Symposium on Principles of Database Systems*, May 1995.

[Kei94] I. Keidar. A Highly Available Paradigm for Consistent Object Replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Israel, 1994.

[KvRvST93] F. M. Kaashoek, R. van Renesse, H. van Staveren and A. S. Tanenbaum. FLIP: an Internetwork Protocol for Supporting Distributed Systems. In *ACM Transactions on Computer Systems*, February 1993.

[KTV93] F. M. Kaashoek, A. S. Tanenbaum and K. Verstoep. Using Group Communication to Implement a Fault-Tolerant Directory Service. In *Proceedings of the IEEE 13th International Conference on Distributed Computing Systems*, pages 130-139, May 1993.

[Lam78] L. Lamport. Time, Clocks, and The Ordering of Events in a Distributed System. *Comm. ACM*, 21(7), pages 558-565. 1978.

[LLSG90] R. Ladin, B. Liskov, L. Shrira and S. Ghemawat. Lazy Replication: Exploiting the Semantics of Distributed Services. In *Proceedings of the 9th Annual Symposium on Principles of Distributed Computing*, pages 43-58, August 1990.

[LLSG92] R. Ladin, B. Liskov, L. Shrira and S. Ghemawat. Providing Availability Using Lazy Replication. *ACM Transactions on Computer Systems,* 10(4), pages 360-391.

[Mac94] R. A. Macedo. Fault-Tolerant Group Communication Protocols for Asynchronous Systems. Ph.D. Thesis, Department of Computer Science, University of Newcastle Upon Tyne, 1994.

[Mal94] D. Malki. Multicast Communication for High Availability. Ph.D. thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Israel, 1994.

[MAMA94] L. E. Moser, Y. Amir, P. M. Melliar-Smith and D. A. Agarwal. Extended Virtual Synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems,* pages 56-65, June 1994. IEEE. A detailed version appears as ECE Technical Report #93-22, University of California, Santa Barbara, December 1993.

[MES93] R. A. Macedo, P. Ezhilchelvan, S. K. Shrivastava. Newtop: a Total Order Multicast Protocol Using Causal Blocks. BROADCAST project deliverable report, Volume I, October, 1993; available from Dept. of Computer Science, University of Newcastle upon Tyne, UK.

[MM93] P. M. Melliar-Smith and L. E. Moser. Trans: A Reliable Broadcast Protocol. *IEE Transactions on Communications*, 140(6), pages 481-493, December 1993.

[MMA90] 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, January 1990.

[MMA91] P. M. Melliar-Smith, L. E. Moser and D. A. Agarwal. Ring-based Ordering Protocols. In *Proceedings of the International Conference on Information Engineering*, pages 882-891, December 1991.

[MMA93] L. E. Moser, P. M. Melliar-Smith and V. Agrawala. Asynchronous Fault-Tolerant Total Ordering Algorithms. In *SIAM Journal of Computing*, 22(4), pages 727-750, August 1993.

[MMA94] L. E. Moser, P. M. Melliar-Smith and V. Agarwala. Processor Membership in Asynchronous Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems* 5(5), pages 459-473, May 1994.

[MPS91] S. Mishra, L. L. Peterson and R. D. Schlichting. A Membership Protocol Based on Partial Order. In *Proceedings of the International Working Conference on Dependable Computing for Critical Applications*, pages 309-331, February 1991.

[PBS89] L. L. Peterson, N. C. Buchholz and R. D. Schlichting. Preserving and Using Context Information in Interprocess Communication. In *ACM Transactions on Computer Systems*, 7(3), pages 217-246, August 1989.

[Pow91] D. Powell, editor. Delta-4 - A Generic Architecture for Dependable Distributed Computing. *Esprit Research Reports*, Springer Verlag, November 1991.

[PL88] J. F. Paris and D. D. E. Long. Efficient Dynamic Voting Algorithms. In *Proceedings of the 4th International Conference on Data Engineering*, pages 268-275, February 1988.

[PL91] C. Pu and A. Leff. Replica Control in Distributed Systems: An Asynchronous Approach. In *ACM SIGMOD International Conference on Management of Data*, pages 377-386, May 1991.

[RM89] B. Rajagopalan and P. K. McKinley. A Token-Based Protocol for Reliable Ordered Multicast Communication. In *Proceedings of the 8th IEEE Symposium on Reliable Distributed Systems*, pages 84-93, October 1989.

[RV92] L. Rodrigues and P. Verissimo. *x*AMp: a Multi-primitive Group Communication Service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, October 1992.

[RVR93] L. Rodrigues, P. Verissimo and J. Rufino. A Low-level Processor Group Membership Protocol for LANs. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 541-550, May 1993.

[Ske82] D. Skeen. A Quorum-Based Commit Protocol. In *Berkeley Workshop on Distributed Data Management and Computer Network*, number 6, pages 69-80, February 1982.

[SS93] Andre Schiper and Alain Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 561-568, May 1993. IEEE.

[Tho79] R. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2) pages 180-209, June 1979.

[vRBFHK95] R. van Renesse, K. Birman, R. Friedman, M. Hayden and D. Karr. A Framework for Protocol Composition in Horus. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, August 1995.