A Highly Available Paradigm for Consistent Object Replication

A thesis submitted in fulfillment
of the requirements for the degree of
Master of Science

by
Idit Keidar

supervised by
Prof. Danny Dolev

Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel.

April 12, 1994

Acknowledgments

I would like to thank my eternally optimistic advisor Prof. Danny Dolev, for his most helpful guidance, for believing in this work and for his encouragement. He carefully read my drafts and assisted in the formulation of the algorithm and proofs.

Yair Amir greatly contributed to this work with his original ideas and insight; his enthusiasm persuaded me to work on this subject. Special thanks to Dalia Malki, who carefully read my drafts, and suggested simplifications to the algorithm and proofs. All three were integral in asking the intriguing questions that helped sharpen the blurry points.

I'd especially like to thank Prof. Catriel Beeri, who provided helpful comments on the first version of this work; his suggestions resulted in a clearer presentation.

Finally, I'd like to thank my husband, Mark, for listening all along and for putting up with "the bugs at midnight".

Abstract

This work provides an efficient paradigm for object replication using the State Machine approach, overcoming network partitions and reconnects. The *Consistent Object Replication Layer* (*COReL*) supplies the application builder with *long-term* services such as reconciliation of states among recovered and reconnected processors and global message ordering. *COReL* is a high-level communication service layer, designed in the Transis environment. It supplies the services defined for the *Replication Service* layer in HORUS and Transis.

We present an algorithm for totally ordering messages in the face of network partitions and site failures. The novelty of this algorithm is that it always allows a majority (or quorum) of connected processors in the network to make progress (*i.e.* totally order messages), if they remain connected for sufficiently long, regardless of past failures. Furthermore, our algorithm always allows processors to initiate messages, even when they are not members of a connected majority component in the network. Thus, messages can eventually become totally ordered even if their initiator is never a member of a majority component. The algorithm orders each message within two communication rounds, if no failures occur during these rounds.

We describe how *COReL* may be used in the design of distributed and replicated database systems. We present an *atomic commitment protocol* (*ACP*) based on *COReL*. The novelty of this ACP is that it always allows a majority (or quorum) of processors that become connected to resolve the transaction, if they remain connected for sufficiently long. We know of no other ACP with this feature. We suggest a paradigm for replica control, based on *COReL*, that always allows a majority of connected processors to update the database.

Contents

1 Introduction

Reliability and availability of loosely coupled distributed systems is becoming a requirement for many installations. The availability of data and services can be increased by replication. If the data is replicated in several sites, it can still be available after site failures, and communication-link failures. However, implementing an object with several copies residing on different sites may introduce inconsistencies between copies of the same object. Consistency may be achieved by applying updates in the same order at all the replica.

We present an algorithm that implements replication using the State Machine approach [29], overcoming network partitions and reconnects. Replica of the state machine receive all the update messages in the same order. If the communication allows, all the replica eventually reach the same state. The algorithm totally orders messages in the face of network partitions and site failures. The algorithm was designed in the Transis [3] environment; Transis is a sophisticated transport layer that supplies group multicast and membership services.

The *Consistent Object Replication Layer* (*COReL*) supplies the application builder with *long-term* services such as reconciliation of states among recovered and reconnected processors and global message ordering. It supports consistent object replication over dynamic networks; the network may partition into several components[1], and remerge. *COReL* is a high-level communication service layer, designed in the Transis environment. It supplies the services defined for the *Replication Service* layer in HORUS and Transis [22]. *COReL* may be implemented over any transport layer that supplies similar group membership and multicast services.

It is well known that reaching agreement in asynchronous environments with a possibility of even one failure is impossible [17]. Every fault-tolerant algorithm that correctly solves the consensus problem in an asynchronous environment is bound to have an infinite run, in which no processor fails and no decision can be made. In the algorithm we present, this difficulty is encapsulated in the membership protocol of the underlying Transport Layer. When we claim that a quorum of connected processors may always totally order their messages, we implicitly require that the underlying membership protocol will not falsely report of failures. In theory, the *adversary* may delay messages in such a way that will cause the membership protocol to constantly change the view. In this case, the presented algorithm might run indefinitely without ordering even a single message. In practice, we expect false-detections to be rare.

We present an algorithm for totally ordering messages in the face of network partitions and site failures. The novelty of this algorithm is that it always allows a majority (or quorum) of connected processors in the network to make progress (totally order messages), if they remain connected for sufficiently long, regardless of past failures. Some previous algorithms allow a majority to continue operating in the face of one failure, but if failures cascade, and later a majority becomes connected, it might remain blocked. Furthermore, our algorithm

[1]A **component** is sometimes called a **partition**. In our terminology, a partition splits the network into several components.

always allows processors to initiate messages, even when they are not members of a majority component. Thus, messages can eventually become totally ordered even if their initiator is never a member of a majority component. The algorithm orders each message within two communication rounds, if no failures occur during these rounds[2].

In Section 8 we describe how *COReL* may be used in the design of distributed and replicated database systems. We suggest to represent transactions as messages, and disseminate them using *COReL*. *COReL* guarantees that all the transactions will reach all the sites in the same order. If the database servers at the different sites are identical *deterministic state machines* this is sufficient to ensure consistency. Otherwise, an *atomic commitment protocol* (*ACP*) is invoked to guarantee transaction atomicity. In Section 8.3 we present an ACP based on *COReL*. The novelty of this ACP is that it always allows a majority (or quorum) of processors that become connected to resolve the transaction, if they remain connected for sufficiently long. We know of no other ACP with this feature. Moreover, when *COReL* is used for transaction dissemination and ordering as well as for atomic commitment, the processors that succeed to resolve the transaction, are also allowed to totally order new messages (*i.e.* to perform updates to the database). Thus, the database is always available to a quorum of connected processors. If an ACP is invoked for each transaction, and partition occurs before the votes of the detached processors are received by the coordinator, the transaction is aborted. In a dynamic network, where view changes are frequent, this may cause many transactions to abort. In our approach, if the servers are deterministic state machines, transactions may become totally ordered and may be applied even if members that did not acknowledge them detach. A network partition never forces us to abort a transaction. Other advantages of our approach are the ability to *pipeline* transactions, and piggyback all the "voting" information on regular messages, and efficient message dissemination that takes advantage of network broadcast capabilities. Furthermore, using *COReL* for all the communication tasks supports simple design of database schedulers, free of communication trouble.

This thesis is organized as follows: Section 2 is a survey of related work, in Section 3 we describe the semantics and guarantees of the provided replication service. In Section 4 we formally describe the model for the environment in which *COReL* may be implemented, *e.g.* the Transis environment. In Section 5 we present the algorithm, and in Section 6 we prove its correctness. In Section 7 we suggest some extensions to the algorithm. These extensions are not presented in full detail, and are brought without proof of correctness. In Section 8 we describe how *COReL* may be used in the design of distributed and replicated database systems. Section 9 concludes the thesis.

[2]By "no failures occur" we implicitly mean that the underlying membership service does not falsely report of failures.

2 Related Work

The *Consistent Object Replication Layer* (*COReL*) is built as a high-level communication service layer, on top of a transport layer that supplies reliable group communication services. *COReL* supplies highly available services for consistent replication, and may be used to design efficient protocols for distributed databases. We survey paradigms for group communication in Section 2.1. In Section 2.2 we survey algorithms for fault tolerant replication that are based on group communication. In Section 2.3 we survey *commit* protocols for distributed transactions, and in Section 2.4 we survey protocols for replication in database systems.

2.1 Group Communication

Communication mechanisms are central to any replicated or distributed system. Today, distributed and replicated database systems are built using point-to-point communication mechanisms, such as TCP/IP, DECNET, ISO and SNA. These mechanisms provide reliable point-to-point message passing between live and connected processors. These mechanisms do not make efficient use of the hardware capabilities. In particular, the broadcast or multicast mechanisms are not used. Replicated systems are natural candidates for using these mechanisms, since each update message is sent to multiple destinations. In addition, a replicated system requires *global order* on updates (or transactions). Group multicast paradigms that use broadcast mechanisms support efficient message ordering, while point-to-point communication does not support global message ordering. Thus, using group communication mechanisms can lead to simpler and more efficient solutions for replication.

One of the leading systems in the area of group communication is the ISIS system [7, 9]. The novelty of ISIS is in the formal and rigorous definition of the service interface. ISIS supplies group communication services that maintain the *virtual synchrony* [8] property, which is important for consistent application semantics. When the network partitions, ISIS allows only one network component (the *primary*) to operate in the system. The members of disconnected components are *blocked*. This approach can lead to inconsistent results if a processor that is extracted from the primary component continues to deliver messages for a "while" before recognizing the partition. The ISIS system is currently being redesigned and built from scratch. The new version of ISIS is called HORUS; it employs a membership algorithm based on the approach of the Transis membership [13], that allows several network components to coexist in case of network partitions.

The AMOEBA distributed operating system supplies group communication services [20]. These services support reliable totally ordered broadcast among process groups. The service is implemented on top of an unreliable network, taking advantage of hardware broadcast capabilities. The membership service of the AMOEBA system lets the user determine the minimal size with which the system can continue operating. If the user determines a majority threshold, the result is a primary-component membership service. On lower thresholds, the system may partition. AMOEBA does not provide any solution for merging operational partitions upon reconnection.

Melliar-Smith et al. suggest in [23] a protocol for reliable broadcast communication over physical LANs, the *Trans* protocol. Similar ideas appear in the Psync protocol [27]. These protocols use the hardware broadcast capability for message dissemination and a combined system of positive and negative acknowledgments to detect message losses and recover them. Melliar-Smith et al. provide the *Total* protocol for total ordering of messages over Trans [25].

The Transis System [3] supports fast and reliable message multicast among currently connected processors, on top of an unreliable network. Transis utilizes the characteristics of available hardware, and recovers message losses using an algorithm similar to the Trans and Psync algorithms. An alternative algorithm for message recovery and total order, Totem [5], is also implemented in the Transis environment. The environment is dynamic and processors can come up and may crash, may partition and re-merge. Transis maintains the *membership* [2] of connected processors automatically. The membership service of Transis supports the virtual synchrony property, and provides consistent semantics as described in [13] that are required for the algorithm we present here (see Section 4).

Recent work by Moser et al. [24] defines a computation model that supports network partitions – the *Extended Virtual Synchrony* (*EVS*) Model. The model rigorously defines the semantics of message multicast in a partitionable network. These semantics are valuable for developing applications that tolerate partitions, such as the algorithm presented here. In the Transis environment, two total ordering algorithms that support EVS [5, 3] are implemented.

2.2 Fault Tolerance and Replication with Group Communication

One of the main applications of group multicast is object replication. Totally ordered multicast guarantees the consistency of all the replica of an object (if the object may be represented as a deterministic state machine). To increase availability when failures are possible, *fault tolerant* algorithms must be designed. Most of the algorithms for group multicast mentioned above, detect failures, and extract faulty members from the membership. Usually, only a *primary component* is allowed to continue operating when a partition occurs. These algorithms do not recover the states of reconnected processors; but rather rely on their application to supply *long-term* services such as reconciliation of states. The only exception that we know of is the Total algorithm [25].

Total does not maintain a membership of the connected processors, and does not *give up* on processors that do not respond for a long time. Total is highly *fault tolerant*, it allows all the processors to send messages, and if a majority of the processors are alive and connected, they eventually succeed to totally order messages with a high probability. Total, however, is too complicated to be of practical use. It incurs a high overhead: even if no failures occur it requires a minimum of five communication rounds to order a message, and the maximum number of communication rounds required is not bounded (but is finite with a probability close to one). The algorithm we present here requires two communication rounds to order a message if no failures occur during these rounds[3]. Thus, our algorithm supplies services

[3]By "no failures occur" we implicitly mean that the underlying membership service does not falsely report

similar to those supplied by Total, and with the same level of fault-tolerance. Using the membership service of Transis, we provide a simpler and more efficient protocol. In Total, all the processors "vote" on the order of each message separately, whereas in our algorithm, the membership service allows us to collect votes only when configuration changes occur. In our algorithm, a connected majority does not need to take into consideration concurrent messages in other components.

The algorithm in [1] exploits the Transis membership to design a replicated server that tolerates network partitions. The novelty of [1] is the idea to separate between message dissemination and message ordering, allowing all the processors to initiate messages, and disseminating them among the processors as fast as the communication allows. The ordering of messages is done separately. The main drawback of this algorithm is that messages can not be ordered without being acknowledged by all the processors. Thus, if one processor crashes, the algorithm does not order new messages until its recovery.

2.3 Distributed Transaction Commitment

A lot of work has been done in the area of distributed and replicated databases. Most of the suggested protocols, as well as most of the existing distributed database systems use an *atomic commitment protocol* (*ACP*): To ensure consistency in distributed databases when a transaction spans several sites, the database servers at all sites have to reach a common decision whether the transaction should be committed or not. A mixed decision results in an inconsistent database, a unanimous decision guarantees the *atomicity* of the transaction (provided that each local server can guarantee local atomicity of transactions). To this end an ACP is invoked. Usually, the ACP is invoked for each transaction separately. Chapter 7 of [6] contains a detailed presentation of atomic commitment protocols.

The simplest and most renowned ACP is *two phase commit* (*2PC*). Several variations of 2PC were suggested, the simplest version is centralized – one of the processors is designated as the *transaction coordinator*. The transaction coordinator sends a transaction (or request to prepare to commit) to all the participants. Each processor answers by a **Yes** ("ready to commit") or by a **No** ("abort") message. If any processor votes No, all the processors abort. The transaction coordinator collects all the responses and informs all the processors of the decision. In absence of failures, this protocol preserves atomicity. Between the two phases, each processor *blocks*, *i.e.* keeps the local database locked, waiting for the final word from the transaction coordinator. If a processor fails before its vote reaches the transaction coordinator, it is usually assumed that it had voted No. If the transaction coordinator fails, all the processors remain blocked indefinitely, unable to resolve the last transaction.

The 2PC protocol is an example of a *blocking* protocol: operational sites sometimes wait on the recovery of failed sites. Locks must be held in the database while the transaction is blocked. Even though blocking preserves consistency, it is highly undesirable because the locks acquired by the locked transaction cannot be relinquished, rendering the data

of failures. In a practical system, fault detectors may be fine-tuned to avoid false detections almost entirely.

inaccessible by other requests. Consequently, the availability of data stored in reliable sites can be limited by the availability of the weakest component in the distributed system.

For this reason, Skeen [30] introduced the notion of *non-blocking* protocols for atomic commitment. A protocol that never requires an operational site to block until a failure is recovered is called a *non-blocking protocol.* Skeen presented a rigorous model of crash recovery in distributed systems [32]. In this model, it is assumed that failures can be detected, and therefore *consensus* is achievable (see Section 4.3), and non-blocking protocols may be designed.

Skeen designed a family of protocols, the three phase commit protocols [30], that are resilient to *site failures.* These protocols consist of a basic three-phase atomic commitment protocol that is invoked while there are no failures, and a special *Termination Protocol* that is invoked when failures are detected. These protocols never require an operational site to block (where "operational site" refers to a site that has not failed since the beginning of the transaction). These protocols are not resilient to network partitions, and may lead to inconsistent results if partitions occur. Skeen et al. proved [32] that there exists no non-blocking protocol resilient to network partitioning. When a partition occurs, the best protocols allow no more than one group of sites to continue while the remaining groups block.

Skeen suggested a variation of three phase commit that maintains consistency in spite of network partitions [31]. This protocol is blocking; it is possible for an operational site to be blocked until a failure is mended. In case of failures, the algorithm uses a *majority* or *quorum* based termination protocol, that allows a majority set (or a quorum) to resolve the transaction. If failures cascade, however, a majority of processors can become connected and still remain blocked. As it was proved that completely non-blocking termination is impossible to achieve, further research in this area concentrated on minimizing the number of blocked sites when partitions occur. Chin et al. [11] define *Optimal Termination Protocols* in terms of the average number of sites that are blocked when a partition occurs. The average is over all the possible partitions, and all the possible states in the protocol in which the partitions occurs. The analysis deals only with states in the original commit protocol, and ignores the possibility for cascading failures (failures that occur during the termination protocol). It is proved that any ACP with optimal termination protocols takes at least three phases, and that the quorum-based termination protocols are optimal. To our knowledge, no ACP was suggested that always allows a connected majority to proceed, regardless of past failures. In Section 8.3 we present a commitment protocol based on *COReL* that always allows a connected majority to resolve the transaction (if it remains connected for sufficiently long). Another advantage of our approach is the ability to *pipeline* transactions. Generally, an ACP is invoked for each transaction separately. Three phase commit protocols involve special rounds of communication, dedicated to exchanging "voting" messages of the protocol, while in our approach, all the information needed for the protocol is piggybacked on regular messages.

2.4 Database Replication

Extensive work was done in the area of database replication. Replication can increase performance, if queries are more frequent than updates, and may increase availability. To increase availability, the *replica control protocol* must be resilient to failures. Network partitions are generally considered the most disruptive type of failures.

A replicated database is considered correct if it behaves as if each object has only one copy, as far as the user can tell. This property is called *one-copy equivalence.* In a one-copy database, the system should ensure *serializability*; that is, interleaved execution of user transactions is equivalent to some serial execution of those transactions. A replicated database system is considered correct if it is one-copy serializable (1SR), *i.e.* it ensures serializability and one-copy equivalence. Chapter 8 of [6] contains a detailed definition of 1SR histories and their representation, and a description of several protocols that maintain 1SR.

Several protocols were suggested for maintaining 1SR in spite of network partitions, allowing a primary component to process transactions while other processors are blocked. The most basic ones are the Primary Site/Copy algorithms. In this approach, one copy of an item is designated as the primary copy, and is responsible for all the operations on that item. In the case of failure, only the partition that contains the primary copy can access the data item. Updates are forwarded at recovery to regain consistency. The drawback of this approach is that if the primary site fails, all the other sites remain blocked.

Other algorithms are based on quorums. The majority consensus approach [33] was generalized to quorum consensus with Gifford's [18] weighted voting algorithm. In this algorithm, each copy of a data item is assigned a number of votes. The user defines a *read-quorum* r and a *write-quorum* w for each item, such that $r + w > v$ and $w > v/2$ where v is the total number of votes assigned to the item. Read operations are required to collect information from r sites, and write operations to write to w sites. The dissemination is per operation, and not per transaction, and as a result, the transaction is blocked after every operation, waiting for response. In our approach the entire transaction executes locally as one unit, without waiting for response on every operation. Herlihy [19] generalized the quorum consensus approach to data types with semantical operations other than read and write. In this approach each data-type defines operations on quorums for these operations. This design also supports dynamic change of the quorums.

To reduce the communication, El Abbadi et al. suggested the Accessible Copies Algorithm [16], and its generalization described in [15]. These algorithms maintain a *view* of the system. Since perfect knowledge is not always possible, the processors maintain an approximate view of the network – a *virtual partition.* A data item can be read/written within a component (virtual partition) only if a majority of its read/write votes are assigned to copies that reside on sites that are members of this component. In this case the data item is said to be *accessible.* Read and write operations on accessible data items are done by collecting sub-quorums of the number of votes in the component. For example, if the read sub-quorum is one vote, read operations are implemented by reading the nearest copy of the item, and write operations are implemented by writing all copies residing on members of the component. The maintenance of virtual partitions greatly complicates the algorithm. When the view changes, the processors need to run a protocol to agree on the new view, and recover the most up-to-date state. Our approach frees the database server of communication tasks, and it needs only concentrate on scheduling of transactions at its local site.

In order to guarantee the atomicity of transactions, the algorithms mentioned above use an ACP, and therefore are bound to block if the coordinator fails in the course of the ACP. Thus, these approaches do not *always* allow a connected majority to update the database. Our approach always allows a quorum to resolve transactions that were initialized in previous components, and continue to update the database. Moreover, if a partition occurs before the votes of the detached processors are received by the coordinator, the transaction is aborted. In a dynamic network, where view changes are frequent, this may cause many transactions to abort. In our approach, if the servers are deterministic state machines, transactions may become totally ordered and may be applied even if members that did not acknowledge them detach. A network partition never forces us to abort a transaction.

Davidson et al. [12] survey approaches to replication in the face of partition failures. They classify partition processing strategies for replicated databases along two orthogonal lines. In the first dimension, they propose that consistency can be sacrificed to increase availability. *Pessimistic* strategies prevent inconsistencies by limiting availability. Each component makes the worst-case assumptions on what other components are doing. All the protocols surveyed above are pessimistic. *Optimistic* strategies do not limit the availability. Any transaction may be executed in any component. If distinct components operate on the same item concurrently, inconsistencies can be introduced. When the partition recovers, the system must detect inconsistencies and resolve them. The results of a committed transaction may need to be undone. *COReL* may be used in the design of optimistic servers, by using *COReL* to disseminate the messages, and applying them to the database before they are ordered. We do not elaborate on this idea in this work.

The second dimension, distinguishes between *syntactic* and *semantic* approaches. Syntactic approaches use 1SR as their sole correctness criterion, and make no limitations on the semantics of the transactions. All the protocols we surveyed above are syntactic. *COReL* is suitable for this approach, because the total order on transactions may guarantee 1SR. Semantic approaches, on the other hand, limit the transaction model. An example of a semantic limitation is the requirement that all operations be *commutative*. In this case, executions that are interleaved in different orders yield identical results. *COReL* may also be used in the design of applications with a semantic approach, in this case the ordering feature of *COReL* is not needed, and *COReL* is used only for guaranteed-delivery message dissemination.

Pu et al. [28] suggest an asynchronous approach to replica control; they introduce the notion of *epsilon serializability* (*ESR*). ESR suggests a tradeoff between availability and consistency, it allows inconsistent data to be seen, but requires that data will eventually converge to a consistent (1SR) state. Several replica control protocols for maintaining ESR were suggested. Some of these limit the transaction model, *e.g.* using only commutative operations.

Another replica control protocol proposed in [28] is the *ordered updates* (ORDUP) protocol. The idea behind ORDUP is to execute transactions asynchronously, but in the same order at all replica. Update transactions are represented as messages and are disseminated to all the sites; they are applied to the database when they are totally ordered. There are no limitations on transaction semantics. We adopt this approach for replication using *COReL*. Their work does not suggest an efficient algorithm for disseminating and ordering messages. We propose that *COReL* be used to perform these tasks efficiently, *i.e.* we suggest to implement the ORDUP replica control service as a *COReL* application.

3 The Service Model

The *Consistent Object Replication Layer* (*COReL*) supplies the application builder with *long-term* services such as reconciliation of states among recovered and reconnected processors and global message ordering. It supports consistent object replication over dynamic networks; the network may partition into several components, and remerge. It is explicitly assumed that multiple network components may exist in the system simultaneously. *COReL* is designed as a high level communication layer, supporting the services defined for the *Replication Service* layer in HORUS and Transis [22]. This layer is built on top of a transport layer that supplies reliable group multicast and membership services with the restrictions described in Section 4.

COReL supports long-term reliable multicast; it guarantees that all the messages will eventually reach all their destinations, and will be totally ordered in the same order everywhere. *COReL* is resilient to both processor failures and communication link failures. The algorithm we present is *non-blocking*; it always allows processors to multicast messages. Furthermore, it always allows members of a connected primary component in the network to make progress (totally order messages).

3.1 *COReL* groups

COReL supports consistent replication services within fixed *replication groups*. A *COReL* group is static and determined by the application. It represents a long-term replication group, which is not changed by transient failures and recoveries. A *COReL* group will usually consist of a set of replicated *servers* (*e.g.* a group of database servers in a replicated database setting). In the sequel, we will restrict our discussion to one *COReL* group of servers.

3.2 General Architecture

A fixed set of servers (*All_Servers*) are running *COReL*. The set of participating processors is known to all servers. Each server uses *COReL* to multicast messages to the other servers. *COReL* is implemented as a library and the server code is linked with it. We refer to the server as the *COReL application*. Each copy of *COReL* receives messages from its application, disseminates them to the other replica of *COReL* and delivers the messages to its application. The *COReL* replica communicate using the Transport Layer. Below, we will refer to *the instance of COReL at site p* simply as processor p.

Each processor maintains a local message queue $\mathcal{MQ}$, that is an ordered list of all the messages that this processor received from the Transport Layer. After message m was received by *COReL* at site p, and p wrote it on stable storage (in its $\mathcal{MQ}$) we say that *p has* the message m. Messages are uniquely identified through a pair $< sender, counter >$. This pair will be referred to as the *message id*.

This work deals with the *ordering* of messages and events. We define the *causal* partial order below, motivated by Lamport's [21] definition of order of events in a distributed system.

Definition: Causal Order The causal order of message delivery is defined as the transitive closure of:

(1) $m \xrightarrow{cause} m'$ if $\text{receive}_q(m) \rightarrow \text{send}_q(m')$

(2) $m \xrightarrow{cause} m'$ if $\text{send}_q(m) \rightarrow \text{send}_q(m')$

3.3 Service Levels

At each server, *COReL* builds its knowledge about the order of messages at other processors. We use the colors model defined in [1] to indicate the knowledge level associated with each message, as described below. The application may request of *COReL* to receive up-calls with each message when the message becomes locally ordered (*red*), when it becomes globally totally ordered (*green*) and/or when it becomes stable (*white*). *COReL* associates colors to each message, and supplies its application with the following service levels:

red: No knowledge about the message's global total order (among All_Servers). A message in $\mathcal{MQ}$ is *red* if there is no knowledge that it has a different color. Red messages have the following guarantees:

- Guaranteed global *causal* order; the red messages are received in an order that is consistent with the causal partial order.
- Guaranteed local total order for new messages in the current network component (*i.e.* all the members of a connected network component receive new red message in the same order).

green: Knowledge about the message's global total order. A processor *totally orders a message in the global total order* when it knows that no other message will be ordered before it. A message becomes *green* when it is totally ordered. The set of green messages at each site at a given time is a prefix of $\mathcal{MQ}$. The last green message in $\mathcal{MQ}$ marks the *green line*, *i.e.* all the messages that precede it are green.

white: Knowledge that all the other processors know the message's order. A message becomes *white* (or *stable*) at processor p, when p knows that every processor in All_Servers has totally ordered it (marked it as green)[4].

The notion of *white* messages is needed only for garbage collection. *COReL* may discard messages when they become white, since no other server will need information about them in the future. For simplicity of the presentation in this paper, we will not distinguish between green and white messages. We will refer to messages that are either green or white simply as *green messages.*

3.4 Message Dissemination

Each copy of *COReL* logs all the messages that it receives from the Transport Layer. When network failures are mended and previously disconnected network components remerge, the logged messages are replayed to members that didn't receive them beforehand. Each processor disseminates all the messages that it received, regardless of their initiator. This way, two processors that never directly communicate with each other, will eventually receive each other's messages from other processors. The algorithm disseminates messages to all the targets as fast as communication allows. This property is called the *Eventual Path Requirement* in [1].

3.5 Message Ordering

When the network partitions into several components, the algorithm always allows a *primary component*, that remains connected for sufficiently long, to continue ordering messages. Processors in other components may continue to send messages but cannot agree upon their place in the total order before communicating with members of a primary component. Messages initiated in minority components may diffuse through the system and become totally ordered even if their initiator is never a member of a primary component.

We use a *quorum system* to decide if a group of connected processors may become a primary component. Different quorums systems may be used, at the user's choice. In Section 4.4 we specify the requirements of the quorum system.

At every point during the history of the protocol, if a quorum of processors become connected, and remain connected for sufficiently long, these processors will manage to proceed and totally order all their messages. By *remain connected* we implicitly assume that the underlying membership protocol does not report of view changes. We elaborate on this assumption in Section 4.3.

[4]Determining when a message is white (or stable) is not a service of the Replication Layer in HORUS and Transis. This service is supplied by a separate *Stability Layer* that is run on top of the Replication Layer.

4 The Environment Model

COReL is a high-level communication service layer, designed in the Transis environment. Transis is a sophisticated transport layer [3, 2] that supplies omission fault free group multicast and membership services with a built-in flow control in an asynchronous environment. *COReL* uses Transis as a building block for reliable communication within connected network components[5]. *COReL* may be implemented using any transport layer that supplies similar services, such as Totem [5], Trans [23] and HORUS [35, 34] with the membership service of Transis [2, 13].

4.1 Group Multicast and Membership

The Transis environment supports multicast communication among *groups of processors*. The basic communication primitive is to post (*send*) a message to a group. Transis *delivers* a posted message to all the members of the group, including the member that sent it. When Transis delivers a message to its application we say that the application *receives* the message. In our discussion, *COReL* is a Transis application.

After a group is created, the group undergoes *configuration* (*membership*) *changes* when processors are added to the group (*e.g.* when they *join* the group, or when detached components of the group reconnect); and when members are taken out of the group (*e.g.* when members voluntarily *leave* the group or when they detach or crash). The membership service of Transis reports these changes to the application through special *configuration change* (*membership change*) messages. Configuration change messages are delivered among the stream of *regular* messages. The first message delivered to any processor when it joins or forms a new group is a configuration-change message containing the set of processors that are currently members of the group. Thus, during the execution of an application in the Transis environment, Transis delivers to it a sequence of regular messages interposed by configuration change messages.

Definition: Let m be a message, such that the last configuration-change message preceding m is C. Then we say that m is delivered in the *context of* C. Or shorter, m is delivered in C.

4.2 Transport Layer's Total Order Properties

COReL may be implemented using any transport layer that supplies reliable locally ordered group multicast and membership services that maintain the following properties:

Property 4.1 *The Transport Layer totally orders the messages within each component. A logical* timestamp (TS) *or serial number is attached to every message when it is delivered. The TS has the following properties:*

[5]A **component** is sometimes called a **partition**. In our terminology, a partition separates the network into several components.

- *The same timestamp is attached to the message at all the processors that deliver it.*
- *Different messages have different timestamps, even if they are delivered in different components.*
- *The TS total order preserves the* causal *partial order, defined by Lamport [21].*

The Transport Layer delivers messages at each site in timestamp order[6]. *This implies that every two messages are delivered in the same order at all the processors that deliver both of them.*

Property 4.2 *Let p and q be processors, and assume that both p and q deliver the same two consecutive configuration changes C_1, C_2. Then for every message m that p delivers between C_1 and C_2, processor q also delivers m between C_1 and C_2.*

When the configuration changes from C_1 to C_2, Property 4.2 guarantees full agreement on the set of messages delivered in the context of C_1 among processors that remain connected in spite of the configuration change.

Among processors that do not remain connected, we would also like to guarantee agreement to some extent. If two processors become disconnected, we do not expect to achieve full agreement on the set of messages they delivered in the context of C_1 before detaching. Instead, we require that they agree on a subset of the messages that they deliver in C_1, as described below. Let processors p and q be members of C_1. Assume that p delivers a message m before m' in C_1, and that q delivers m', but without delivering m. This can happen only if p and q became disconnected (from Properties 4.1 and 4.2, they will not both be members of the same next configuration). In Property 4.3 we require that if q delivers m' without m, then no message m'' sent by q, after delivering m', can be received by p in the context of C_1.

Property 4.3 *Let p and q be members of configuration C. If p delivers a message m before m' in C, and if q delivers m' and later sends a message m'', such that p delivers m'' in C, then q delivers m before m'.*

These properties are fulfilled in the Agreed Communication service of the *Extended Virtual Synchrony (EVS) model* [24]. Therefore, *COReL* may be implemented using any transport layer that supports EVS.

In the Transis environment, three different protocols for Total Ordering of messages are implemented. The protocol in [3] and Totem [5] support the services of the EVS model.

[6]This property is required for messages at the level of the transport layer. If *COReL* retransmits the same application message, the retransmission is regarded as a new transport layer message.

4.3 Correctness of the Membership Service

We expect the membership service to *correctly* reflect the network situation, and we expect connected processors to have the same view of the membership in the system. Since it is well known that reaching agreement in asynchronous environments is impossible [17], this goal may not be achieved in any practical implementation of a membership protocol. Existing membership protocols circumvent this difficulty in different ways. The protocols in [2, 5, 13] use an inaccurate failure detector, based on timeout: when a processor is presumed *faulty*, it is taken out of the view. Messages from processors that are not part of the current view are *discarded*. A presumed failed machine can re-join the membership. In practice, failure detection can be fine-tuned to avoid false detection almost entirely.

In the algorithm we present, this difficulty is encapsulated in the membership protocol. When we claim that a quorum of connected processors may always totally order their messages, we implicitly require that the underlying membership protocol will not report of any false detections. In theory, the *adversary* may delay messages in such a way that will cause the membership protocol to constantly change the view. This way, the presented algorithm can run indefinitely without ordering even a single message.

Consensus can be achieved, however, in an asynchronous system with *external fault-detectors* as described in [10]. If we strengthen the model of the underlying membership algorithm to work with external fault detectors, we can guarantee that a quorum of connected processors will eventually succeed in ordering messages.

The algorithm is *correct* in both models, in the sense that it does not order messages differently at different sites. The *liveness* of the algorithm (its ability to make progress), depends on the underlying transport layer's behavior.

4.4 Quorums and Primary Components

When the network partitions into several components, the algorithm allows a *primary component* to continue ordering messages. Processors in other components may continue to send messages but cannot agree upon their place in the total order before communicating with members of a primary component.

We use a *quorum system* to decide if a group of connected processors may become a primary component. To enable maximum flexibility we allow the quorum system to be elected in a variety of ways (*e.g.* majority). We assume a predicate $Quorum(S)$ that is TRUE for a given subset S of the processors iff S is a quorum. The requirement from this predicate is that for any two sets of processors S and S' such that $S \cap S' = \emptyset$, at most one of $Quorum(S)$ and $Quorum(S')$ holds, *i.e.* every pair of quorums intersect. Numerous quorum systems that fulfill these criteria were suggested. An analysis of the availability of different quorum systems may be found in [26].

5 The Algorithm

We present an algorithm for guaranteed delivery and total ordering of multicast messages. The algorithm is symmetric, all the participating processors run the same protocol.

The algorithm guarantees that all messages will eventually reach all sites, and will be totally ordered (become green) in the same order at all sites. It always allows members of a connected primary component to totally order messages. The algorithm is resilient to both processor failures and network partitions.

5.1 Guaranteed Delivery

When processors fail or when the network partitions, messages are disseminated in the restricted context of a smaller configuration, and are not received at sites which are members of other components. The participating processors keep these messages for as long as they might be needed for retransmission, *i.e.* until they become white. Each processor logs (on stable storage) every message that it receives from the Transport Layer. A processor acknowledges a message after it is written to stable storage. The acknowledgments (*ACKs*) may be piggybacked on regular messages.

When network failures are mended and previously disconnected network components remerge, a Recovery Procedure is invoked; the members of the new configuration exchange messages containing information about messages in previous components and their order. If they remain connected for sufficiently long, each processor knows exactly which messages every other member has. They now determine which processor will retransmit which message (in case retransmissions are needed), and the logged messages are replayed to members that didn't receive them beforehand.

5.2 Message Ordering

Within each component messages are ordered by the Transport Layer. The Transport Layer supplies a unique timestamp[7] (*TS*) for each message when it delivers the message to *COReL*. We use this order to determine the global total order. When *COReL* receives the message, it writes the message on stable storage along with its TS.

When a message is retransmitted, the timestamp that was given when the original transmission of the message was received, is attached to the retransmitted message, and is the only timestamp used for this message.

Incoming messages within each component are inserted at the end of the local $\mathcal{MQ}$, thus $\mathcal{MQ}$ reflects the order of the messages local to this component. When components merge, retransmitted messages from other components are inserted into the queue in an order that may interleave with local messages (but never preceding green messages).

We use the notion of a *primary component* to allow members of one network component to continue ordering messages when a partition occurs. For each processor, the *primary*

[7]This timestamp is globally unique, even in the face of partitions.

component bit is set iff this processor is currently a member of a primary component. In Section 5.6.1 we describe how the primary component is maintained. Messages that are received in the context of a primary component (*i.e.* when the primary component bit is set) may become totally ordered according to the following rule:

Order Rule 1 *Members of the current primary component PM are allowed to totally order a message (in the global order) once the message was acknowledged by all the members of PM.*

Note: Messages may become green by Order Rule 1 in contradiction to the TS order (with respect to causally concurrent messages in other components).

If a message is totally ordered (*i.e.* becomes green) at some processor p according to this rule, then p knows that all the other members of the primary component received the message, and have written it on stable storage. Furthermore, the algorithm guarantees that all the other members already have an obligation to enforce this decision in any future component using the following mechanism:

We define an intermediate, *yellow*, level of knowledge between the red and the green, among members of a primary component. Yellow is not a new service level, it is only an internal indication used by *COReL*. Each processor marks as yellow messages that it received and acknowledged in the context of a primary component, and as a result, might have become green at other members of the primary component. A processor marks a message as green when it knows that all the other members of the primary component know that the message is yellow. For each processor, the yellow messages are the next candidates to become green.

Yellow messages precede all the red messages in $\mathcal{MQ}$, and follow all the green messages. Thus, $\mathcal{MQ}$ is divided into three zones: a green prefix, then a yellow zone and a red suffix. The last yellow message in $\mathcal{MQ}$ marks the *yellow line*; the set of messages between the yellow and the green lines are yellow.

When components merge, members of the last primary component enforce all the green and the yellow messages that they have before any concurrent red messages. Concurrent red messages from distinct components are interleaved according to the TS order.

Thus, if a member of a primary component marks a message as green according to Order Rule 1 then all the other members of the primary component have the message marked as yellow or green. Since two primary components always intersect, in the next primary component that will be formed there will be at least one member that is obliged to this message (*i.e.* has it marked as yellow or green). This member will make sure that this message will precede any concurrent message from a minority component.

5.3 Notation

We use the following notation:

- $\mathcal{MQ}^p$ is the $\mathcal{MQ}$ of processor p.

- $\mathit{Prefix}(\mathcal{MQ}^p, m)$ is the prefix of $\mathcal{MQ}^p$ ending at message m.
- At each processor, $\mathcal{MQ}$ imposes an order on the set of messages in it. We denote by $m \overset{p}{\prec} m'$ the fact that m precedes m' in $\mathcal{MQ}^p$.
- $\mathit{Green}(\mathcal{MQ}^p)$ is the green prefix of $\mathcal{MQ}^p$.
- We define *processor p knows of a primary component PM* recursively as follows:
  1. If a processor p was a member of PM then p *knows* of PM.
  2. If a processor q *knows* of PM, and p recovers the state of q, then p *knows* of PM.

5.4 Invariants of the Algorithm

Both red and green messages are delivered to the application in an order that is consistent with the *causal* partial order defined by Lamport [21]. The order of messages in $\mathcal{MQ}$ of each processor always preserves the causal partial order.

Messages that are totally ordered are marked as green. Once a message is marked as green, its place in the total order may not change, and no new message may be ordered before it. Therefore, at each processor, the order of green messages in $\mathcal{MQ}$ is never altered. Furthermore, the algorithm totally orders messages in the same order at all sites, therefore the different processors must agree on their green prefixes.

The following properties are *invariants* of the algorithm; they are maintained by each step of the algorithm:

FIFO If a processor p has in its $\mathcal{MQ}$ a message m that was originally sent by processor q, then for every message m' that q sent before m, $\mathcal{MQ}^p$ contains m' before m.

Causal If a processor p has in its $\mathcal{MQ}$ a message m that was originally sent by processor q, then for every message m' that q had in its $\mathcal{MQ}$ before sending m, $\mathcal{MQ}^p$ contains m' before m.

No Changes in Green New green messages are appended to the end of $\mathit{Green}(\mathcal{MQ}^p)$, and this is the only way that $\mathit{Green}(\mathcal{MQ}^p)$ may change.

Agreed Green The processors have compatible green prefixes: for every pair of processors p and q running the algorithm, and for every $\mathit{Green}(\mathcal{MQ}^p)$, (at every point in the course of the algorithm), and every $\mathit{Green}(\mathcal{MQ}^q)$, one of $\mathit{Green}(\mathcal{MQ}^p)$ and $\mathit{Green}(\mathcal{MQ}^q)$ is a prefix of the other.

Yellow If a processor p marked a message m as green in the context of a primary component PM, and if a processor q *knows* of PM, then:

1. Processor q has m marked as yellow or green.
2. $\mathit{Prefix}(\mathcal{MQ}^q, m) = \mathit{Prefix}(\mathcal{MQ}^p, m)$.

In Section 6 we formally prove that these invariants hold.

5.5 Handling Messages

COReL is event driven. Each *COReL* server reacts to messages as it receives them:

- When *COReL* receives a *message from the application* (a *down-call*):
  Create a *Regular Message* that reflects this application message and disseminates it among the servers using the Transport Layer[8].

- When a *Regular Message* m is received from the Transport Layer:

  - Insert m to $\mathcal{MQ}$, and write it on stable storage. If the primary component bit is set, mark the message as yellow.
  - Send an ACK[9] for m. (It is possible to piggyback the ACK on the next message that will be sent).
  - If the *red* service level is requested, notify the application of the new "red" message. Note: The application does not distinguish between red and yellow messages. Yellow is not a service level, it is only an internal indication used by *COReL*.

- When an ACK is received (possibly piggybacked on a regular message):
  If the primary component bit is set, try to totally order messages (*i.e.* mark messages as green) according to Order Rule 1. When a message is totally ordered, notify the application of the new green message.

- The handler for *Configuration Change Messages* is described in the next section.

5.6 Handling Configuration Changes

The main subtleties of the algorithm are in handling configuration changes. Faults can occur at any point in the course of the protocol, and the algorithm ensures that even in the face of cascading faults, no inconsistencies are introduced. To this end, every step taken by the handler for configuration changes must maintain the invariants described in Section 5.4.

When merging components, messages that were passed in the more restricted context of previous components need to be disseminated to all members of the new configuration. Green and yellow messages from a primary component should precede messages that were concurrently passed in other components. All the members of the new configuration must agree upon the order of all past messages. To this end, the processors run the Recovery Procedure.

[8]The message is not be inserted into $\mathcal{MQ}$ until it is delivered by the Transport Layer to *COReL*.

[9]The Stability Layer in HORUS and Transis may be used for disseminating and analyzing ACKs. *COReL* may run on top of an instance of the Stability Layer, and receive an up-call whenever a message is *locally stable*, *i.e.* acknowledged by all the members of the current configuration. In this case *COReL* will send the ACK to the stability layer, and the stability layer will disseminate it.

If the new configuration C introduces new members, the Recovery Procedure is invoked in order to bring all the members of the new configuration to a common state. New messages that are delivered in the context of C are not inserted into $\mathcal{MQ}$ before the Recovery Procedure, and thus the **FIFO** and **Causal** invariants are not violated. The members of C exchange *state messages*, containing information about messages in previous components and their order. In addition, each processor reports of the last primary component that it knows of, and of its green and yellow lines. Every processor that receives all the state messages knows exactly which messages every other member has, the messages that not all the members have are retransmitted.

In the course of the Recovery Procedure, the members agree on the green and yellow lines. The new green line is the *maximum* of the green lines of all the members; every message that one of the members of C had marked as green, becomes green for all the members. The id of the last green message is sufficient to denote the set of green messages for each processor, since the order of green messages is globally agreed (as stated by the **Agreed Green** invariant).

The members that *know* of the latest primary component, PM, determine the new yellow line. The new yellow line is the *minimum* of the yellow lines of the members that know of PM. If some message m is red for a member that knows of PM, then by the **Yellow** invariant, it was not marked as green by any member of PM. In this case if any member had marked m as yellow, it changes m back to red. The members that determine the new yellow line all know of the same latest primary component, and since new messages are only marked as yellow in the context of primary components, there are no contradictions in the yellow zones of these processors, their green and yellow prefixes are all prefixes of some agreed order. Therefore, the id of the latest yellow message is sufficient to denote the set of yellow messages at these processors. Note that in general, yellow zones of different processors, that know of different primary components, are not always prefixes of some agreed global order. A detailed description of the Recovery Procedure is presented in Section 5.6.2.

After reaching an agreed state, the members of a majority component in the network may practice their right to totally order new messages. They must order all the yellow messages first, before new messages, and before red messages form other components, in order to be consistent with decisions made in previous primary components.

If for a new configuration C, $Quorum(C)$ holds, the members of C will try to establish a new primary component. The algorithm for establishing a new primary component is described in Section 5.6.1. All committed primary components are sequentially numbered (as we prove in Claim 4). We refer to the primary component with sequential number i as PM_i.

In the course of the run of the configuration change handler, the primary component bit is unset, regular messages are blocked, and no new regular messages are initiated. The handler for configuration changes is described in Figure 1.

Configuration Change Handler for the Configuration C:

- Unset the primary component bit.
- Stop handling regular messages, and stop sending regular messages.
- If C contains new members, run the Recovery Procedure described in Section 5.6.2.
- If $Quorum(C)$ holds, run the algorithm for establishing a new primary component, described in Section 5.6.1.
- Continue handling and sending regular messages.

Figure 1: Configuration Change Handler

5.6.1 Establishing a Primary Component

A new configuration, C, is established as the new primary component, if $Quorum(C)$ holds, after the retransmission phase described in Section 5.6.2. The primary component is established in a three-phase agreement protocol. In the first phase all the processors declare of their intention to establish the new primary component. In the second phase, the members **commit** to establish the new primary component, and mark all the messages in their $\mathcal{MQ}$ as yellow. In the **establish** phase, all the processors mark all the messages in their $\mathcal{MQ}$ as green. A processor marks the messages in its $\mathcal{MQ}$ as green only when it knows that all the other members marked them as yellow. Thus, if a failure occurs in the course of the protocol, the **Yellow** invariant is not violated.

The new primary component is established as follows:

- All the members *attempt* to establish C as the new primary component.
- After receiving attempt messages from all the members, each processor *commits* to the new primary component, and marking all the non-green messages in its $\mathcal{MQ}$ as yellow.
- After receiving commit messages from all the members, each processor *establishes* the new primary component by setting the primary component bit to TRUE, and marks all the messages in its $\mathcal{MQ}$ as green. (These messages should now become green according to Order Rule 1).
- If the Transport Layer reports of a configuration change before the process is over – the establishing is aborted, but none of its effects are undone. The primary component bit remains unset until the next successful establish process.

Each processor maintains the following variables:

Last_Committed_Primary is the number of the last primary component that this processor has committed to establish.

Last_Attempted_Primary is the number of the last primary component that this processor has attempted to establish. This number may be higher than the number of the last component actually committed to, in the case of failures.

The algorithm for establishing a new primary component is described in Figure 2.

Establishing a New Primary Component
If C contains new members, the Recovery Procedure is run first. If $Quorum(C)$ holds all members of a configuration C try to establish it as the new primary component:

- Compute the number of the new primary being established:
  $New\_Primary = \max_{i \in C}(Last\_Attempted\_Primary_i) + 1$.
  The members of C now try to establish it as $PM_{New\_Primary}$.
- Attempt to establish $PM_{New\_Primary}$:
  Set *Last_Attempted_Primary* to $New\_Primary$ on stable storage, and send an attempt message, to notify the other members of the attempt.
- Wait for attempt messages from all members of C. When these messages arrive, do the following in one atomic step:
  1. Commit to the configuration by setting $Last\_Committed\_Primary$ to $New\_Primary$ on stable storage.
  2. Mark all the messages in the $\mathcal{MQ}$ that are not green as yellow.

  Send a commit message, to notify the other members of the commitment.
- Wait for commit messages from all members of C, and then *establish* C, by setting to TRUE the primary component bit. Mark as green all the messages in $\mathcal{MQ}$.
- If the Transport Layer reports of a configuration change before the process is over – the establishing is aborted, but its effects are not undone.

Figure 2: Establishing a new primary component

5.6.2 Recovery Procedure

If the new configuration, C, introduces new members, then each processor that delivers the configuration change runs the following protocol:

Recovery Procedure for processor p

1. Send state message including the following information:
   - $Last\_Committed\_Primary$.
   - *Last_Attempted_Primary*.

- The id of the last message that p received from every processor q. Note that this is sufficient to represent the set of messages that p has, because the order of messages in $\mathcal{MQ}^p$ always preserves the *causal* order, and in particular, the messages from each processor arrive in FIFO order.
- The id of the latest green message (green line).
- The id of the latest yellow message (yellow line).

2. Wait for state messages from all the other processors in the new configuration. The state messages reflect the set of messages each processor had at the time of the configuration change.

   From the state messages it can be determined which messages should be retransmitted and which processors have these messages.

3. Compute:
   $Max\_Committed = \max_{p \in C} Last\_Committed\_Primary_p$.

   Denote as *Representatives* all the members that have $Last\_Committed\_Primary = Max\_Committed$.

   The *Representatives* advance their green lines to include all messages that any member of C had marked as green.

   The *Representatives* retreat their yellow lines to include only messages that all of them had marked as yellow, and in the same order: if processor p has a message m marked as yellow, while another member with $Last\_Committed\_Primary = Max\_Committed$ has m marked as red, or does not have m at all, then p changes to red m along with any messages that follow it in $\mathcal{MQ}^p$.

4. A unique representative from the group of *Representatives* is chosen deterministically.

   If all the members have the same last committed primary component, (*i.e.* all are *Representatives*), go directly to Step 7, no representative needs to be chosen.

   Determine the following sets of messages:

   component_stable is the set of messages that all the members in the new configuration have.

   component_ordered is the set of messages that were ordered by every member in the new configuration (*i.e.* messages that are green for all the members of C).

   priority are yellow and green messages that the representative has.

5. Retransmission of priority messages:

   The chosen representative computes the maximal prefix of its $\mathcal{MQ}$ that contains only component_ordered messages. It sends the set of priority messages in its $\mathcal{MQ}$ that follow this prefix. For component_stable messages, it sends only the header (including the original ACKs), and the other messages are sent with their data and original piggybacked ACKs.

Members from other configurations insert these messages into their $\mathcal{MQ}$s, in the order of the retransmission, following the green prefix, and ahead of any non_priority messages.

Note that it is possible for members to already have some of these messages, and even in a contradicting order (but in this case, not as green messages). In this case they adopt the order enforced by the representative.

6. If $Last\_Committed\_Primary_p < Max\_Committed$; do the following in one atomic step:

   - If p has yellow messages that were not retransmitted by the representative, change these messages to red, and reorder them in the red part of $\mathcal{MQ}$ according to the TS order.
   - Set *Last_Committed_Primary* on stable storage to *Max_Committed.*
   - Set the green and yellow lines according to the representative; the yellow line is the last retransmitted message.

7. Retransmission of red messages:

   Messages that not all the members have, need to be retransmitted. Each message is retransmitted by at most one processor; this can be determined according to any deterministic rule. The processors that need to retransmit messages send them, with their original ACKs, in an order maintaining the Retransmission Rule (described in Figure 3).

Retransmission Rule *If processor p has messages m and m' such that m' is ordered after m in p's messages queue, then during Step 7 of the Recovery Procedure:*

- *If p has to retransmit both messages then it will retransmit m before m'.*
- *If p has to retransmit m' and another processor q has to retransmit m then p does not retransmit m' before receiving the retransmission of m.*

Figure 3: Retransmission Rule

   Concurrent retransmitted messages from different processors are interleaved in $\mathcal{MQ}$ according to the TS order of their original transmissions.

After receiving all of the retransmitted messages: if $Primary(C)$ holds then the members try to establish a new configuration. (The algorithm is described Section 5.6.1).

Note: If the transport layer reports of a configuration change before the protocol is over, the protocol is immediately restarted for the new configuration. The effects of the non-completed run of the protocol do not need to be undone.

If the configuration change reports only of processor faults, and no new members are introduced, the processors need only establish the new configuration and no retransmissions are needed. This is due to the fact that, from Property 4.2 of the Transport Layer, all the members received the same set of messages until the configuration change.

6 Correctness Proof of the Algorithm

We now prove the correctness of the algorithm. In Section 6.1 we prove that the order of messages in $\mathcal{MQ}$ of each processor always preserves the causal partial order, and thus, the total order determined by the algorithm at each site preserves the causal partial order. We conclude that at each site messages become both red and green in causal order.

In Section 6.2 we prove that messages are totally ordered in the same order at all the processors. The proof is based on the order imposed on committed primary components. We prove that if a message m is marked as green by some processor p in the context of some primary component, then in all the later primary components, all the members will agree with p on m's order. In other words, every primary component preserves the order determined by previous primary components.

Notation

We denote by p **commits** j the event that p, as a member of PM_j, commits to PM_j when trying to establish it as the new primary component. We denote by p **adopts** j the event that in Step 6 of the Recovery Procedure p sets its $Last\_Committed\_Primary$ to j according to the representative's $Last\_Committed\_Primary$.

We denote by $C_p(C)$ the event that processor p receives the Configuration Change message introducing the configuration C.

The Epochs Model

The processors running the protocol may be viewed as state machines; they react to messages that they receive by the Transport Layer. The *event* $e(m,p)$ is the reaction of processor p when it receives the message m. The event may include internal state changes as well as transmission of messages by p.

A *history* of the protocol is a set of events, partially ordered by the *causal* partial order. In Section 3 we define the causal order of messages motivated by Lamport's [21] definition of the order of events in a distributed system. We generalize the definition of causal order to events and configurations as follows:

- The **causal order of events** is defined as follows:
  $e(m,p) \xrightarrow{cause} e(m',q)$ if $m \xrightarrow{cause} m'$.

- The **causal order of configurations** is defined as follows: Configuration C causally precedes configuration C' if $(\exists p)(\exists q)\ C_p(C) \overset{cause}{\longrightarrow} C_q(C')$.

A history is divided into *epochs*:

Definition:

- An event $e(m,p)$ *happens in* $epoch_i(p)$ if e occurs when $Last\_Committed\_Primary_p = i$ and e does not change $Last\_Committed\_Primary_p$.

- The event in which $Last\_Committed\_Primary_p$ is changed to i is the first event in $epoch_i(p)$. Note that $Last\_Committed\_Primary_p$ may change in two types of events, when p commits i, or when p adopts i.

We say that $epoch_i(p)$ *is empty* if in the history of p, $Last\_Committed\_Primary_p$ was never i.

6.1 Causal Order

We first prove that the order determined by the algorithm at each site preserves the causal partial order.

Claim 1 *Messages are received at each site in an order preserving the causal partial order, and the order of the messages in $\mathcal{MQ}$ of each processor always preserves the causal partial order.*

Proof: Messages are ordered at each processor's $\mathcal{MQ}$ when they are first delivered to it. This order may be altered only in the course of a configuration change protocol when a representative of a primary component enforces its order over the TS order. The proof is by induction on the steps of the protocol in which a message is transmitted or reordered in $\mathcal{MQ}$.

When a regular message is first transmitted to the members of the current configuration we assume that the Transport Layer delivers it in TS order, (which preserves the causal order), and without missing causally preceding messages (in the context of the same configuration). The messages are inserted into each $\mathcal{MQ}$ in this order.

In a new configuration, C, regular messages are sent only after the Recovery Procedure ends, and therefore are received by each processor after all the messages that were sent in configurations that causally precede C.

We now show that during a configuration change the order in which retransmitted messages are delivered at each site, and placed in $\mathcal{MQ}$ preserves causality.

We assume (by induction) that for each member p of a configuration C that receives the configuration change message introducing C, the order of messages in $\mathcal{MQ}^p$ preserves the causal partial order when the configuration change occurs, and we show that this property still holds throughout the Recovery Procedure.

By the Retransmission Rule the retransmission order preserves the order of the messages in $\mathcal{MQ}$ of the retransmitting processor. From our assumption, this order preserves the causal partial order. Therefore, from the assumption on the transport layer, the delivery order of retransmitted messages preserves the causal partial order.

Retransmitted messages are inserted into $\mathcal{MQ}$ according to the TS order with the exception of one special case: messages that are retransmitted by the chosen representative, r, in Step 5 of the Recovery Procedure are inserted into $\mathcal{MQ}$ at the receiving end ahead of non_priority messages, *i.e.* before any messages that r didn't have marked as yellow or green. This does not violate causality, since, from the inductive assumption on $\mathcal{MQ}^r$, the non_priority messages do not causally precede any priority message in $\mathcal{MQ}^r$. Thus, in all cases, retransmitted messages are inserted into $\mathcal{MQ}$ in causal order.

If $m' \overset{p}{\prec} m$ and p reorders m to follow m' in Step 5 of the Recovery Procedure then the representative, r, either had m' and not m, or m' before m. Therefore, by the inductive assumption on $\mathcal{MQ}^r$, m' does not causally follow m. $\Box$

Corollary 2 *Red messages are delivered to the application in an order that preserves the causal partial order.*

Proof: Follows from Claim 1 and the fact that messages become red in the order that they are inserted into $\mathcal{MQ}$. $\Box$

Corollary 3 *The total order of messages computed at each site extends the causal order.*

Proof: Follows from Claim 1 and the fact that the ordered messages are a prefix of $\mathcal{MQ}$. $\Box$

6.2 Total Order

We now prove that messages are totally ordered in the same order at all the processors. The proof is based on the order imposed on committed primary components. We prove that if a message m is marked as green by some processor p in the context of some primary component, then in all the later primary components, all the members will agree with p on m's order. In other words, every primary component preserves the order determined by previous primary components. The proof is by induction on the committed primary components.

Claim 4 *If p and q committed to primary components with number i, PM_i^p and PM_i^q respectively, then PM_i^p and PM_i^q are the same.*

Proof: Assume the contrary. Since every two primary components intersect there is a processor r that is a member of both PM_i^p and PM_i^q. Since both configurations were committed to, r attempted to establish both. W.l.o.g. r attempted to establish PM_i^p before attempting to establish PM_i^q, then when trying to establish PM_i^q, r had at least i as the number of the *Last_Attempted_Primary*, and therefore the *New_Primary* suggested for the new configuration is greater than i, which contradicts the assumption. □

Henceforth we will refer to *primary component number* i as PM_i.

Claim 5 *If p adopts j then there exists a processor q s.t. q commits j.*

Proof: A processor sets its *Last_Committed_Primary* to j either when committing to j, or when adopting from another processor that has its *Last_Committed_Primary* set to j. Therefore, there must be one processor that commits to j, in order to *start* the chain. □

Claim 6 *For each processor p, the value of $Last\_Committed\_Primary_p$ does not decrease.*

Proof: In the protocol, a processor may change its *Last_Committed_Primary* in two cases:

- In Step 6 of the Recovery Procedure, when adopting the value of the representative's *Last_Committed_Primary*. In this case the *Last_Committed_Primary* of p does not decrease, otherwise p would have been chosen as the representative.
- When committing to a new primary component PM_j. Assume that immediately before committing to PM_j $Last\_Committed\_Primary_p = i$. We consider two cases:
  - If p committed to i, then before committing to PM_i, p attempted i, therefore, $Last\_Attempted\_Primary_p \geq i$ when $C_p(PM_j)$ occurs.
  - Otherwise, p adopted i, and from Claim 5, some member q of PM_i has committed to PM_i. Since q committed to PM_i all the members of PM_i have attempted to to establish it, and, furthermore, all the members of PM_i set their *Last_Attempted_Primary* to i causally before any processor committed to i. Since every two primary components intersect, there exists a processor r that is a member of both PM_i and PM_j; r's attempt to establish PM_i causally precedes the event that p sets its *Last_Committed_Primary* to i. Therefore, when r starts to run the Recovery Procedure in which p commits to PM_j, $Last\_Attempted\_Primary_r \geq i$, and this is the value that r sends in the state message.

  In both cases, the number j, of the new primary that the members try to establish $> i$.

□

Corollary 7 *If events e and e' happen at p in epochs i and i' respectively, and if $i < i'$ then e happens before e'. Note: the order on e and e' is well defined since they both happen at processor p.*

Proof: Immediate from Claim 6. □

Claim 8 *If processor p totally orders a message m according to Order Rule 1 in the context of PM_i then all the members of PM_i have committed to PM_i. Therefore, epoch i is not empty for these processors.*

Proof: In order to totally order m according to Order Rule 1 in the context of PM_i, p has to establish PM_i (and set to TRUE the primary component bit). Before establishing PM_i, p waits for all the members of PM_i to send a commit message, therefore all the members of PM_i have committed to PM_i. □

We now proceed to the main part of the proof. In Claims 10 through 14 we prove, by induction, that the following proposition holds in $epoch_j(q)$ for each processor q and for all j:

Proposition 9 for processor q in epoch j:
Assume that a message m was marked as green by some processor p_m in the context of PM_{i_m} according to Order Rule 1, and that i_m is the first primary component in which some processor marked m as green according to Order Rule 1. Then:

- *If $j > i_m$ and if $epoch_j(q)$ is non-empty: q has m marked as yellow or green, and* $\text{Prefix}(\mathcal{MQ}^q, m) = \text{Prefix}(\mathcal{MQ}^{p_m}, m)$ *in $epoch_j$.*
- *If $j = i_m$, and if $epoch_j(q)$ is non-empty then starting at a certain point in $epoch_j(q)$, q has m marked as yellow or green and* $\text{Prefix}(\mathcal{MQ}^q, m) = \text{Prefix}(\mathcal{MQ}^{p_m}, m)$.

Definition:
We say that *there are no contradictions in the green zone when Configuration C is introduced* if :

- For each member p of configuration C, $C_p(C)$ occurs.
- Let *Representatives* be the set of members of C with the highest *Last_Committed_Primary* (as denoted in Step 3 of the Recovery Procedure). For each member p of C, and for each message m that p has marked as green in $\mathcal{MQ}^p$ before $C_p(C)$ occurs:
  - All the *Representatives* have m marked as yellow or green.
  - Processor p agrees with all the *Representatives* on the ordered prefix of its $\mathcal{MQ}$ that ends at m.

Claim 10 *Assume that* there are no contradictions in the green zone when the Configuration C is introduced. *Let m be a message that all the* Representatives *have as yellow or green and agree on* $\text{Prefix}(\mathcal{MQ}, m)$ *when they receive the Configuration Change Message introducing C.*

Then, at the end of Step 7 of the Recovery Procedure all the members of C that execute Step 7 have identical $\mathcal{MQ}$s, and the message order in their Prefix$(\mathcal{MQ}, m)$ *is the same as the order in* Prefix$(\mathcal{MQ}, m)$ *of the Representatives when $C_p(C)$ occurred. The message order in the $\mathcal{MQ}$s is not altered until the end of this instance of the Recovery Procedure.*

Proof: We first show that all the members agree on a prefix of their $\mathcal{MQ}$s that contains all the messages that were marked as green by at least one of the members before this instance of the Recovery Procedure.

Let m be a message that all the *Representatives* have as yellow or green and agree on *Prefix*$(\mathcal{MQ}, m)$ when they receive the Configuration Change Message introducing C.

In Step 3 of the Recovery Procedure all the *Representatives* mark m as yellow or green, and the order of messages in their $\mathcal{MQ}$s is not altered. At the end of this step, all the *Representatives* agree on the prefix of green and yellow messages in their $\mathcal{MQ}$s. From the assumption, this prefix is consistent with the order in the green prefixes of all the other members of C.

In Step 5 of the Recovery Procedure there are two cases to consider:

- If m is component_ordered, *i.e.* m is green for all the members of C, then, from the assumption, all the members have identical prefixes ending at m. In this case, m is not retransmitted, m remains green and its order isn't altered.

- Otherwise, m (or its header) is retransmitted by the representative. Since no member had green messages in an order that contradicts *Prefix*$(\mathcal{MQ}, m)$ of the *Representatives*, all the members adopt the order of the representatives on a prefix of their $\mathcal{MQ}$s that contains m.

In Step 6 all the members of C mark m as green or yellow, (according to the representative's color) At the end of this step, all the members have the same set of green and yellow messages in their $\mathcal{MQ}$s, and in the same order.

We have shown that at the end of Step 6 all the members of C that execute this step agree on a prefix of their $\mathcal{MQ}$s that contains all the messages that were marked as green by one of the members before this instance of the Recovery Procedure. Furthermore, they agree on the prefix of green and yellow messages in their $\mathcal{MQ}$s. We now show that at the end of Step 7 all the members of C that execute Step 7 have identical $\mathcal{MQ}$s. And indeed, before this step they agree on the order of all green and yellow messages. During this step, all the members retransmit all the red messages, and insert them into their $\mathcal{MQ}$s. Thus, all the processors have the same set of red messages in their $\mathcal{MQ}$s. The red messages in each $\mathcal{MQ}$ are ordered according to the timestamp order of their original transmissions. Therefore, at the end of this step, all the members of C have identical $\mathcal{MQ}$s, and the message order is consistent with the order in the *Prefix*$(\mathcal{MQ}, m)$ of the *Representatives* when $C_p(C)$ occurred.

No new messages arrive until the end of the Recovery Procedure, and the message order in each $\mathcal{MQ}$ is not altered until the end of this instance of the Recovery Procedure. □

Claim 11 *Assume that for each member p of the configuration C, $C_p(C)$ occurs (*i.e. *p receives the configuration change message introducing C), and let j be the highest value of $Last\_Committed\_Primary_p$ among members of C at the time $C_p(C)$ occurs. If Proposition 9 holds for each member p of C for every epoch $i \leq j$ when $C_p(C)$ occurs, then throughout this execution of the Recovery Procedure, Proposition 9 holds for all the members of C for every epoch $i \leq j$.*

Proof: Let m be a message that was first marked as green according to Order Rule 1 in the context of PM_{i_m} (by some processor p_m) , and assume that $j > i_m$. From the assumption, all the *Representatives* (members with $Last\_Committed\_Primary = j$) have m as yellow or green when they start to run the protocol, and their $Prefix(\mathcal{MQ}, m)$ are identical to $Prefix(\mathcal{MQ}^{p_m}, m)$. Therefore, in Step 3 of the Recovery Procedure, none of the members changes m to red, and all the messages in their $Prefix(\mathcal{MQ}, m)$ are yellow or green.

Furthermore, if any other member p of C has m marked as green, then $Prefix(\mathcal{MQ}^p, m) = Prefix(\mathcal{MQ}^{p_m}, m)$

In Step 5 of the Recovery Procedure there are two cases to consider:

- If m is component_ordered, *i.e.* m is green for all the members of C, and all the members have identical prefixes ending at m, then m is not retransmitted, m remains green and its order isn't altered.
- Otherwise, m (or its header) is retransmitted by the representative in Step 5, and marked as green or yellow (according to the representative's color) in Step 6 of the protocol, by every member that sets its $Last\_Committed\_Primary$ to j. No member changes it to red at this step, and no member had it marked as green in contradicting order. Thus, at the end of this step of the Recovery Procedure all the members have m as yellow or green and their $Prefix(\mathcal{MQ}, m)$ are identical.

We have shown that m is not changed to red by any of the members of C in the course of the Recovery Procedure. Therefore, if for a processor p, $Last\_Committed\_Primary_p \geq i_m$ when p started to run the protocol, Proposition 9 still holds for p at any point in the course of the protocol.

It is now left to show for the case that $Last\_Committed\_Primary_p$ is initially smaller than i_m, but is changed in the course of the protocol. In this case, p adopts j in Step 6 of the Recovery Procedure.

And indeed, if p sets its $Last\_Committed\_Primary$ to j on stable storage in Step 6 of the protocol, then, from the discussion above p had already received m before Step 6 of the protocol, and in Step 6, p marks it as green or yellow, according to its color at the representative, and adopts the order of messages in $Prefix(\mathcal{MQ}, m)$ of the representative, when changing its $Last\_Committed\_Primary$ to j.

Every processor that reaches the end of Step 6 of the Recovery Procedure, has its $Last\_Committed\_Primary \geq i_m$, and has m marked as yellow or green, and its $Prefix(\mathcal{MQ}, m)$ is identical to $Prefix(\mathcal{MQ}^{p_m}, m)$. This is not altered in later steps of the protocol. □

Claim 12 *If at a certain point in $epoch_j(q)$, q has m marked as yellow or green and* $\text{Prefix}(\mathcal{MQ}^q, m) = \text{Prefix}(\mathcal{MQ}^{p_m}, m)$*, then Proposition 9 holds for q in $epoch_j$,* i.e. *this property is not altered in later steps of the protocol.*

Proof: The color of messages may change from yellow to red only in the course of the Recovery Procedure, and the order of messages in $\mathcal{MQ}$ may only be altered in the course of the Recovery Procedure. Therefore, it is sufficient to show that Proposition 9 is invariant under the Recovery Procedure, *i.e.* that if it holds for all the members when they start to run the Recovery Procedure, then it also holds for all the members throughout the execution of the Recovery Procedure.

If not all the members of C receive the configuration change message introducing C, then not all the members of C run the Recovery Procedure, therefore the other members of C receive another configuration change message without going past Step 2 of this instance of the protocol. In this case, no messages are changed to red and the $\mathcal{MQ}$s are not altered.

Therefore, we may restrict our discussion to instances of the Recovery Procedure for configurations C s.t. all the members of C receive the configuration change message that introduces C. The conclusion in this case is derived from Claim 11. □

Claim 13 *If Proposition 9 holds for all the processors in every epoch k s.t. $k < j$, and if q commits j, then Proposition 9 holds for q in epoch j.*

Proof: If q commits j then q is a member of PM_j, and all the members of PM_j have attempted to establish it, therefore all of them received the Configuration Change message introducing PM_j, and all of them reached the end of Step 7 of this instance of the Recovery Procedure.

Let k be $\max_{p \in PM_j} Last\_Committed\_Primary_p$ when $C_p(PM_j)$ occurs. For every message m that one of the members p has green, there exists a first primary component PM_{i_m} in the context of which m was marked as green according to Order Rule 1, and $i_m \leq k$. From the assumption, all the *Representatives* (members with $Last\_Committed\_Primary = k$) have m marked as yellow or green, and agree with p on $Prefix(\mathcal{MQ}, m)$. Therefore, there are no contradictions in the green zone when PM_j is introduced.

Let m be a message that was first marked as green (according to Order Rule 1) in the context of PM_{i_m} by some processor p'. We first show that at a certain point in $epoch_j(q)$, q has m marked as yellow or green and $Prefix(\mathcal{MQ}^q, m) = Prefix(\mathcal{MQ}^{p_m}, m)$. We consider three cases:

- If $j > i_m$, since every two primary components intersect, there exists at least one processor that is a member of both PM_j and PM_{i_m} and from Claim 8, this processor committed to PM_{i_m}, therefore, $k \geq i_m$.

  Let *Representatives* be the following set:
  $\{p \in PM_j : Last\_Committed\_Primary_p = k$ when $C_p(PM_j)$ occurs $\}$.

From the assumption Proposition 9 holds for all the representatives in k, thus, Proposition 9 holds for them when $C_p(PM_j)$ occurs. Therefore, all the representatives have m marked as yellow or green and the prefixes of their $\mathcal{MQ}$s that end at m are identical when $C_p(PM_j)$ occurs. Furthermore, none of the other members have green messages in contradicting order.

Therefore, from Claim 10, at the end of Step 7 of the Recovery Procedure all the members of PM_j have identical $\mathcal{MQ}$s, and the message order is consistent with the order in $\mathit{Prefix}(\mathcal{MQ}^{p'}, m)$ when $C_{p'}(PM_j)$ occurred. The $\mathcal{MQ}$s are not altered until the end of this instance of the Recovery Procedure.

- If $j = i_m$ and some member, p, of PM_j had m in its $\mathcal{MQ}$ when $C_p(PM_j)$ occurred, then m is marked as green by the members of PM_j that *establish* it.

  Every processor that completes the execution of this instance of the Recovery Proceduread *establishes* PM_j marks as green all the messages in its $\mathcal{MQ}$. If some processor *establishes* PM_j then all the other members committed to j, and marked all these messages as yellow. From the above discussion, their $\mathcal{MQ}$s are identical at this point.

- We have shown that for every processor that runs this instance of the Recovery Procedure, Proposition 9 holds in the course of the run of the protocol, and when the protocol ends.

  It is left to show that if m is marked as green according to Order Rule 1 in the context of PM_j by some processor p_m, and p_m first received m in the context of PM_j, then q has m marked as yellow or green at some point in epoch j and $\mathit{Prefix}(\mathcal{MQ}^{p_m}, m) = \mathit{Prefix}(\mathcal{MQ}^{q}, m)$. And indeed, if p_m marked m as green then it received acknowledgments for it from all the members of PM_j, all of them received m in the context of PM_j, and therefore, marked it as yellow when receiving it. From Property 4.3 of the Transport Layer, all the members of PM_j received the same set of messages following the configuration change and before m, and in the same order. These messages are ordered in the $\mathcal{MQ}$ of each processor in the order they are received.

We have shown that in all cases, at a certain point in $\mathit{epoch}_j(q)$, q has m marked as yellow or green and $\mathit{Prefix}(\mathcal{MQ}^{q}, m) = \mathit{Prefix}(\mathcal{MQ}^{p_m}, m)$, it is left to show that this property *persists*, *i.e.* that it is not altered in later steps of the protocol. Claim 12 concludes the proof. □

Claim 14 *Proposition 9 holds for every processor q in every epoch j.*

Proof: The proof is by induction on j:

For $j = 0$, the epoch_0 at each processor precedes committing to any primary component, therefore for all m, $i_m > j$ and Property 4.1 trivially holds.

We now assume that Property 4.1 holds for all the processor in every epoch k s.t. $k < j$, and prove that it holds for all the processors in epoch_j. If no processor has committed to

PM_j, then for all the processors $epoch_j$ is empty and Proposition 9 trivially holds. Otherwise, let q be a processor s.t. $epoch_j(q)$ is non-empty. There are two cases to consider:

- If q commits j - then from Claim 13, Proposition 9 holds for q in epoch j.
- If $epoch_j(q)$ is non-empty and q does not commit to j, then q adopts j in the course of a run of the Recovery Procedurefor some configuration C. In this case, $j = \max_{p \in C}$ $Last\_Committed\_Primary_p$ at the time $C_p(C)$ occurs, and q adopts j in Step 6 of the Recovery Procedure. Since q passed Step 2 of this instance of the protocol, we conclude that for each member p of the configuration C, $C_p(C)$ occurred (*i.e.* p received the configuration change message introducing C).

  Therefore, from Claim 11, Proposition 9 holds for all the members of C for every epoch $i \leq j$, throughout this execution of the Recovery Procedure. In particular, Proposition 9 holds at some point in $epoch_j(q)$. Claim 12 concludes the proof.

□

Corollary 15 *Messages become green in the same order at all sites.*

Proof: From the protocol, the order of green messages in $\mathcal{MQ}$ is never altered. The proof immediately follows from Claim 14. □

7 Extensions

In this section we present two extensions of the algorithm, optimizing it for highly unreliable networks, where a majority of the processors are rarely connected at once. We present these extensions without full details, and without proof. We intend to further develop these ideas in future work.

7.1 Agreeing on Total Order with Order Proposals

In networks where the communication is not reliable, it is possible that a majority component will not be formed for a long period. This may happen even if all the processors are alive, and messages from all of them eventually reach all the destinations. In this case, we would like to enable the processors to totally order messages according to an alternative mechanism, that would not require a primary component to be established.

This mechanism allows one designated processor, the *source*, to issue *Order Proposal* messages. An order proposal *prp* is a suggestion to totally order a defined set of messages, *Msgs*(*prp*). The idea to agree upon the order of a set (or *cut*) of messages is based on the concept introduced in [4].

7.1.1 Initiation of an Order Proposal

The order proposals are initiated by the source and disseminated to all the other processors. Each processor that receives the proposal votes either *Accept* or *Reject* on it, as described below. If All_Servers vote accept on the proposal, it is *applied*, and *Msgs*(*prp*) are totally ordered (marked as green). Otherwise – it is rejected.

The source initiates order proposals only when it is not a member of a quorum component; it does not initiate a new order proposal before learning the result of its previous proposal. The source may initiate a new proposal when it learns that its previous proposal was accepted by all the servers, or that it was rejected by some server.

An order proposal is rejected by a processor p if p has new information about the last committed primary component that was not known to the source when the proposal was initiated. Therefore, an order proposal succeeds only if no primary component is formed after the last committed primary that the source knows of (when initiating the message), by members that didn't yet receive the proposal.

Note that this mechanism will succeed to totally order messages only if none of the processors actually fails, and the message is eventually propagated to all of the processors. Therefore, relying on a single source to initiate order proposals does not affect the resiliency.

We present the concept of order proposals, and how to incorporate them into the algorithm presented in Section 5. The presentation is not rigorous, and the algorithm is not described in full detail. In the future, we intend to prepare a detailed presentation of the subject, including a proof of correctness.

7.1.2 The Structure of an Order Proposal

The source generates Order Proposals and disseminates them to the other processors. Each processor maintains a data structure *Last_Order_Proposal* that contains the last order proposal that it received, and all the information that it has about votes of other processors on the proposal. We do not go into detail in describing the accurate structure of an order proposal. Conceptually, order proposals should contain the following information:

- A unique *Proposal Id* generated by the source.
- $Last\_Committed\_Primary_{prp}$ is the $Last\_Committed\_Primary$ that was known to the source when the proposal *prp* was initiated.
- An ordered list of messages in the source's yellow zone.
- A proposed *Cut*. The cut is a vector containing the maximal message id from each processor that was known to the source when the proposal was initiated. When initiating an Order Proposal, *prp*, the source proposes to totally order the set of yellow and red messages that it has in its $\mathcal{MQ}$ at the time of proposal. We denote this set by *Msgs*(*prp*). The proposed order for red messages is the TS order, and the suggested order for yellow messages is explicitly mentioned in *prp*.

- The green line of the source when the proposal was initiated. (The green line is the id of the last message that the source has green).
- A *status vector* containing the vote of each processor on the proposal. The vote may take 3 values: Accept Reject or Unknown. The *local status* of a proposal is the vote of this processor. When the message is initiated, the value of the source's vote is Accept, and all the other votes are Unknown. The message collects votes as it is propagated to all the servers.

7.1.3 Handling Order Proposals

Upon receiving a new proposal, each processor votes on it and incorporates it into the local data structure.

The general outline of the handler for order proposals is:

1. Check if new Proposal has the same proposal id as *Last_Order_Proposal*, and its local status is not reject. If yes – incorporate new votes into the status vector. If any processor voted reject, set the local status to reject.

   Otherwise, if the new proposal is later than *Last_Order_Proposal*:

   - Save the new message as *Last_Order_Proposal*, annulling any previous commitment to an order proposal.
   - If one of the follows happen:
     - Local *Last_Committed_Primary* is higher than the one in the message.
     - Local *Last_Committed_Primary* is the same as the one in the message, and the message contains a yellow message that you have as red.
     - The status vector contains a reject vote.

     Then – reject the message: Mark the message as rejected. Send a rejection message (the proposal message with your status as reject).
   - Otherwise: Accept the message: Mark the message as accepted. Send an acknowledgment (the proposal message with your status as accept).

2. If the proposal was accepted by *All_Servers*, reorder messages in $\mathcal{MQ}$ according to the proposed order, and mark all the proposed messages as green.

7.1.4 Order Proposals and Configuration Changes

The handling of Order Proposals is incorporated in several steps of configuration changes:

- In Step 1 of the Recovery Procedure, each processor attaches to its state message the *Last_Order_Proposal* that it knows of.

- At the end of Step 2 each processor tries to handle the new order proposals. Since the order proposals are sent outside their causal order, processors may receive a proposal *prp* before recovering *Msgs*(*prp*), or when their *Last_Committed_Primary* is smaller than $Last\_Committed\_Primary_{prp}$. Each processor p responds to each new order proposal, *prp*, as follows:

  - If $Last\_Committed\_Primary_p = Last\_Committed\_Primary_{prp}$ and p has all of *Msgs*(*prp*), p handles *prp* as described in the previous section.
  - If p has ground to reject *prp*, it immediately sends a rejection message.
  - Otherwise, p defers its acceptance until the end of the Recovery Procedure.

  Note: If it is known to the members of the new configuration that *All_Servers* have accepted *prp*, then the members of the new configuration totally order all of *Msgs*(*prp*) before Step 3 of the Recovery Procedure.

- Each processor handles all the deferred order proposals before trying to establish a new primary component, after Step 7 of the Recovery Procedure. A processor that attempts to establish this primary component has already recovered all the messages from the other servers; each processor must vote accept or reject on the proposal before or with the attempt message.

  If all the members of a newly formed primary component have accepted the last order proposal, then they first order messages according to this proposal. In this case the members do not immediately mark the proposed messages as green, the messages become green only when the new primary component is established.

  At the commit phase each processor already knows the votes of all the other members. If the message was accepted by all the members of the current primary component, the processor reorders messages in its $\mathcal{MQ}$ according to the proposal as part of the commit phase. When committing to the new primary component, all the proposed messages become yellow. When the primary component is established, all the messages become green.

7.2 Dynamic Voting for Primary Components

The algorithm presented in Section 5 uses a *quorum system* to decide if a group of processors may become a primary component. We give the user the flexibility of choosing the quorum system. The predicate $Quorum(S)$ is TRUE for a given subset of the processors S iff S is a quorum. The requirement from this predicate is that for any two sets of processors S and S' such that $S \cap S' = \emptyset$, at most one of $Quorum(S)$ and $Quorum(S')$ holds, *i.e.* every pair of quorums intersect. Numerous quorum systems that fulfill these criteria were suggested. An analysis of the availability of different quorum systems may be found in [26].

Many of the proposed algorithms for data replication use quorum systems of some kind, *e.g.* the majority consensus algorithm [33], generalized to quorum consensus with weighted voting by Gifford [18]. Another example is the primary site/copy algorithms that use the *singleton* quorum system, where a group is a quorum iff it contains the primary site.

The concept of quorums can be further generalized, to allow more flexibility yet. El Abbadi et al. [14] suggest the *dynamic voting* paradigm for electing a primary component. This paradigm defines quorums adaptively: when a partition occurs, if a majority of the previous quorum is connected, a new and possibly smaller quorum is chosen. Thus, each primary component must contain a majority of the previous one, but not necessarily a majority of the processors. The drawback is that there can be situations where almost all of the processors are connected, but cannot perform updates, because of the potential existence of past surviving quorum at the processors that are down. To prevent this situation, a lower bound may be defined on the size of primary components. Thus, majority consensus is a private case of dynamic voting, with the minimum primary component size chosen to be - $\lceil n/2 \rceil$. In Section 7.2.1 we formally define the requirements from a *dynamic quorum system*, and from a dynamic paradigm for choosing primary components. In Section 7.2.2 we show how to modify the algorithm described in Section 5.6.1 to support dynamic quorums.

7.2.1 Requirements of a Dynamic Quorum System

A *dynamic quorum system* specifies for each possible group of processors, which of its subsets can become sub-quorums. We assume a predicate $Sub\_Quorum(S, S')$ that is TRUE for $S' \subset S$, iff S is a *sub-quorum* of S. We extend the definition for arbitrary sets, S and T: $Sub\_Quorum(S, T)$ is TRUE iff $Sub\_Quorum(S, T \cap S)$ is TRUE. Note that it is possible that $|T| > |S|$, in this case the new quorum is larger than the previous one. Let Min_Quorum be the minimum quorum size allowed in the system (determined by the user).

The requirements from this predicate are:

- For any two subsets S' and S'' of a set of processors S, such that $S' \cap S'' = \emptyset$, at most one of $Sub\_Quorum(S, S')$ and $Sub\_Quorum(S, S'')$ holds.
- If $|T| < Min\_Quorum$, then for every group S, $Sub\_Quorum(S, T)$ is FALSE.

For the algorithm to be correct, we require a total order on all the committed primary components. When using a static quorum system, this was guaranteed by the intersection property: "every two primary components intersect". Unfortunately, dynamic quorum systems do not possess this property. Instead, we totally order the committed primary components by extending the partial order defined on components that do intersect.

Let PM and PM' be two primary components. If $j \in PM \cap PM'$, and j **commits** to both of these primary components, then at site j one of PM and PM' is committed before the other, w.l.o.g, PM is committed to first. We denote this relation by: $PM \prec PM'$. This relationship is a partial order on the set of committed primary components in the system. The requirement from a dynamic paradigm for choosing primary components is that the transitive closure of $\prec$ will be a total order. In the next section we explain how an algorithm that satisfies this requirement may be constructed.

7.2.2 Using a Dynamic Quorum System for Primary Components

It is possible to adjust the algorithm for establishing a primary component described in Section 5.6.1 for dynamic quorum systems. In this version of the algorithm, each processor needs to maintain not only the number of the latest primary component known to it, but also the *membership* of this primary component. This is required because a new configuration C can become a primary component only if it contains a sub quorum of the previous configuration. This requirement involves a delicate point: If a member p of the configuration PM **commits** to PM and fails before establishing it, it cannot know if another member has **established** it before disconnecting or not. Therefore, it must take both possibilities into account when trying to establish the next primary component. To this end, if p incurs a failure after the commit phase of the establishment protocol, it maintains two "last primary components": the last established primary (*Last_Primary*) that was valid before this run of the protocol, and the *Pending_Committed*, that p committed to but did not establish. If consecutive failures occur, *Pending_Committed* may include more than one configuration.

Note: *Pending_Committed* is non-empty for a processor p only if p receives a configuration change message after committing to a new primary component, but before establishing it. We expect this situation to be very rare.

Before trying to establish the new primary component, all the members must agree on the *Last_Primary* and *Pending_Committed* that need to be considered. This should be done in the course of the Recovery Procedure, and we do not elaborate on it in this paper.

A configuration C may be installed as the new primary component only if:

- $Sub\_Quorum(Last\_Primary, C)$.
  and
- $(\forall S \in Pending\_Committed) Sub\_Quorum(S, C)$.

We do not go into details, nor do we prove the correctness of this algorithm in this paper. We simply describe the concept. We intend to provide a detailed presentation of this algorithm in the future.

8 Using *COReL* to Replicate Databases

The design of replicated databases involves a great deal of communication. Database servers at different sites need to communicate constantly to reach common decisions in order to maintain the consistency of all replica. Efficient communication services are, therefore, the key to designing efficient replicated database systems. *COReL* may supply a variety of services that require communication:

- Each transaction is initiated at one site and must be disseminated to all other sites. In existing systems, this is mostly done per operation, each operation in the transaction is disseminated as part of its execution.

- In distributed databases when a transaction spans several sites the database servers at all sites have to reach a common decision whether the transaction should be committed or not. To this end they run an *atomic commitment protocol* (*ACP*) such as *two phase commit* (*2PC*). Usually, the ACP is invoked for each transaction separately.
- Replicated databases are generally considered *correct* if their schedules maintain *one copy serializability* (*1SR*), this means that transactions execute at all sites in an order equivalent to a serial execution on one copy. In order to ensure 1SR the servers at different sites need to agree on a common order of the transactions.

Usually these tasks are performed separately, thus, several messages need to be passed for each transaction. First, the transaction needs to be disseminated, this involves several messages when different operations of one transaction are transmitted separately. Later an ACP is invoked. 2PC requires two rounds of communication, and a total of $O(n)$ messages, where n is the number of participating sites.

It is possible to perform these tasks together, since agreement on the order of transactions is sufficient for atomic commitment in a replicated database. The database schedulers may be designed in such a way to guarantee that all the sites execute the same transactions in the same order, and all have the same information for decision whether a transaction should be committed or not. Similarly, agreeing on the order of transactions may be used to guarantee 1SR. We explicitly assume that the local database servers at the different sites are identical *deterministic state machines*. Furthermore, in our approach, *read only* transactions execute locally at the site they are initiated. Therefore, the database schedulers must never abort update transactions because of conflicting read only transactions. With these assumptions, the order imposed on transactions is sufficient to guarantee atomic commitment and 1SR, and an ACP does not need to be invoked. In Section 8.1 we present an algorithm for replicating database servers that fulfill these assumptions. If these assumption cannot be fulfilled, an ACP is indeed required. In Section 8.3 we show how an ACP can be constructed on top of *COReL*.

We use *COReL* to disseminate transactions and agree on their order. Thus, each transaction need only be multicast once. All the information needed for decisions is piggybacked on regular messages. The volume of the communication is significantly decreased when the same messages are used for all tasks. Furthermore, transactions may be *pipelined* using *COReL* to achieve greater transaction throughput. *COReL* supports a highly available, fault tolerant replicated database.

8.1 Replicating Database Servers

Transactions may be initiated at all sites. Transactions that update the database are represented as messages and are disseminated using *COReL*. A transaction may be executed when it is totally ordered with respect to other transactions. *COReL* guarantees that all transactions will reach all the processors, and will be executed in the same order.

Read only transactions may be executed locally, immediately when they are submitted, to increase data availability. The increased availability is achieved at the expense of slightly limiting the correctness: Users may see a not up-to-date view of the database, *e.g.* if a user issues an update transaction and later queries the database, the query may actually be executed in the local database before the update transaction. Thus, the database will not reflect the new update, and the user will see an old (but consistent) database state. Many applications will prefer to tolerate this limitation on the semantics in order to increase the availability. If this level of inconsistency is not acceptable, the application builder may chose to disseminate the read only transactions along with the update transactions via *COReL*, and execute them only when they become totally ordered. With the latter approach, no inconsistencies are introduced, but members of minority components in the network are not able to query the database. In the sequel, we do not elaborate on the latter approach; we restrict our discussion to the former.

The local server at the initiating site handles the transaction as follows:

1. If the transaction is *read only* execute it.
2. Otherwise, pass a message with the transaction to the *COReL*. Do not execute it yet. *COReL* will disseminate it to all sites.
3. When *COReL* marks the message as green, it delivers it to the server for execution.
4. The local servers may use any one copy concurrency control mechanism to interleave local transactions, as long as they guarantee an execution equivalent to execution of the transactions in the order determined by *COReL*. In case of deadlocks, the local servers must all make the same decision in choosing the transaction to be aborted. These conditions are fulfilled if the local servers are deterministic state machines.
5. The local schedulers should never abort an update transaction because it conflicts with a *read only* transaction.

This scheme requires transactions to be represented by messages. A common representation of transactions is as series of database operations (read/write). If the transaction involves computation in order to decide what database operations should be executed, then the message may contain the computing program. Often in replicated databases the application code is also replicated, in this case a transaction may be represented by a program name (or several programs) that need to be run, and a list of parameters. Each site runs the program locally. Modern database systems support this option with a mechanism called *stored procedures*. A stored procedure is an implementation of application logic that is run by the database server. Stored procedures support an *object-oriented* design of database systems: the *methods* for each object are stored procedures. The objects are accessed only through the defined methods.

Viewing each transaction as a single message is not actually necessary. The same mechanism may be used for dissemination of single operations, or parts of transactions. As mentioned before, if the local servers are deterministic state machines, consistency will be maintained. This way, interactive systems may be well supported: if a user wishes to *lock* an item, an attempt for lock acquisition can be represented as one message. Since all the messages are totally ordered, lock requests will be handled in the same order at all sites, and therefore if a transaction succeeds to acquire a lock at one site, it succeeds to acquire it at all the sites. The drawback of this approach is that if failures occur, sites may be blocked in the middle of a transaction.

8.2 Synchronous or Asynchronous ?

The above algorithm best supports an asynchronous design of database systems. The transaction is not executed immediately, but only after it is totally ordered with respect to other transactions. The user does not get immediate response as to whether his transaction was committed or not, and the database is not modified yet to reflect the transaction.

The same algorithm may be used to design a synchronous application, by blocking the user until the transaction (or the operation in case the message contains an individual operation) is totally ordered. This is generally the approach of atomic commitment protocols.

The advantage of the asynchronous approach is that users are never blocked, they can continue to initiate transactions even in a minority configuration, where they may not execute transactions. The disadvantage is that the user gets no response as to whether his transaction is committed or not, and the database does not immediately reflect the change.

We can bridge the two approaches by giving the user an answer when possible, and otherwise giving him the answer *maybe*.

- If the local site is a member of a primary component it is possible to reply whether the transaction was committed or aborted within a reasonable time limit (not longer than the delay in 2PC), and the database will immediately reflect the change. Similarly, a user wishing to edit an item will get a response whether the lock acquisition was successful within a reasonable time limit.

- If the server is a not member of the primary component then the user will get the response *maybe*. The transaction will be executed only after the partition will be repaired, and then it will be known whether it will commit or not. Meanwhile, the database remains unchanged. Lock acquisitions will be blocked (or aborted).

- The members of a minority partition may construct a local *partially correct* view of the database with the *maybe* transactions, by applying unordered (*red*) transactions.

This approach greatly increases the database availability especially if network partitions are frequent. Transactions may eventually become totally ordered (*green*) even if their initiator is never a member of a primary component. To decrease the blocking induced by lock acquisitions, it is preferable to use this scheme with *timestamp* based concurrency control, rather than two phase locking, as described in [33]. A description of timestamp based concurrency control may be found in Chapter 4 in [6].

8.3 A Distributed Commitment Protocol Based on *COReL*

In this section we show that total order is sufficient for atomic commitment. We show how to construct an atomic commitment protocol (ACP) that uses *COReL* as a building block. In the protocol we present a majority of connected processors is never blocked, regardless of past failures. We know of no other ACP with this feature.

The ACP we present is constructed like 2PC, but uses totally ordered multicast instead of point-to-point messages. The total order allows us to safely continue when partitions occur. If two different coordinators will be elected in separate network components, and will disseminate different suggestions for a decision (*i.e.* COMMIT and ABORT), these suggestions will be received by all the sites in the same order. All the sites will decide on the first suggestion they will receive.

In a distributed database system, each transaction, T, may span several sites. We denote by $Sites(T)$ the set of sites in which T is executed. For the sake of simplicity, we assume that $Sites(T) = All\_Servers$, *i.e.* that the *COReL* group consists exactly of $Sites(T)$[10]. Every member p of $Sites(T)$ starts running the ACP for T when it receives T for execution (or when it receives a request to prepare to commit T from the initiator of T). Each member votes **Yes** if it may commit T, or **No** if T should be aborted. The votes are multicast to all the members (using *COReL*).

If no failures occur, all the processors receive each other's votes. The decision may be derived from these votes (COMMIT if all the processors voted Yes, and otherwise – ABORT). The member of $Sites(T)$ with the smallest process id is elected as the transaction coordinator, when it receives all the votes it disseminates a *suggestion* for decision using totally ordered multicast. Every member that receives this suggestion, COMMITS or ABORTS according to it.

If a processor receives a configuration change message (for configuration C) before receiving all the votes, a *termination protocol* is started. The member of $Sites(T)$ with the smallest process id in C is elected as the new coordinator (for this network component). Unless cascading membership changes occur, all of the members receive the configuration change. Each member knows if it is the new coordinator or not, so no special messages need to be exchanged. The new coordinator disseminates a new suggestion for decision. If it knows that all the members of $Sites(T)$ voted Yes, the suggestion is COMMIT, and otherwise – ABORT.

Suggestions are disseminated using totally ordered multicast, *i.e.* the green service level. Each processor COMMITS or ABORTS according to the first suggestion that it receives. *COReL* ordered multicast incurs a delay of one communication round before ordering a message. The votes, however, do not need to be totally ordered, the red service level is sufficient. Thus, the presented algorithm involves three communication rounds before reaching a decision. The algorithm is described in Figure 4.

It is easy to see that this protocol fulfills the requirements of atomic commitment (as

[10]We do not address the case of a partially replicated database with separate replication groups for different items in this work.

The Atomic Commitment Protocol for processor p:
Assume that p is a member of the configuration C.

- When the transaction T (or its last operation) is green and delivered by *COReL*, send your vote on it.
- When votes were received from $Sites(T) \cap C$, if p is the member with the smallest process id in the current configuration, p sends a *suggestion* for a decision to all the other members. The suggestion is determined as follows:
  - If votes were received from $Sites(T)$ and if all the sites voted Yes, the suggestion is COMMIT.
  - Otherwise, the suggestion is ABORT.
- When a suggestion arrives, if it is the first suggestion received for this transaction, decide according to it. Otherwise, ignore it.

Figure 4: The Atomic Commitment Protocol

defined in Chapter 7 of [6]):

- All the processors that reach a decision reach the same one. This is achieved by the total order we impose on suggestions: all the sites receive the suggestions in the same order, and decide according to the first one.
- A process cannot reverse its decision after it has reached one.
- The COMMIT decision can only be reached if all processes voted Yes.
- If there are no failures and all processes voted Yes, then the decision will be to COMMIT.
- At any point in the execution of the protocol, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.

8.4 General Database Approach

We presented a novel approach for database replication based on efficient communication. Two paradigms were suggested. The first is a paradigm for replicating deterministic state machines, based on message ordering. With this approach, no atomic commitment protocol is needed, and the replication is achieved efficiently.

In Section 8.3 we presented a protocol for atomic commitment based on total order. This protocol is suitable for distributed and partially replicated database settings.

Our approach has the following advantages:

- Efficient communication, taking advantage of hardware broadcast capabilities.

- Transactions are pipelined.
- The protocol incurs a low overhead.
- The protocol may be implemented using a simple database server, free of communication trouble.
- The delay using our algorithm is no longer than with 2PC, because we only need to wait until all processors acknowledge the message once. (Or if an ACP is needed, it is no longer than the delay in three phase commit).
- When communication or site failures occur, a primary component of connected processors will not be blocked.
- It was shown (see [32]) that in any ACP that tolerates network partitions, there is a time during the execution of the protocol when occurrence of communication failure causes some sites to block. Furthermore, in 2PC up to $n-1$ connected sites may be blocked, if only the coordinator failed. With our algorithm, the primary component service may be designed to guarantee that a majority ($\lceil n/2 \rceil$ connected processors) is never blocked. This is always the case, even if in the history of the system there was a time when a primary component did not exist. To our knowledge, no ACP with this blocking level was previously suggested.
- When *COReL* is used for transaction dissemination and ordering as well as for atomic commitment, the processors that succeed to resolve the transaction, are also allowed to totally order new messages (*i.e.* to perform updates to the database). Thus, the database is always available to a quorum of connected processors. We know of no previous replica control protocol with this feature.

9 Conclusions

We presented an efficient algorithm for totally ordered multicast in an asynchronous environment, that is resilient to network partitions and communication link failures. The novelty of the algorithm is that it always allows a majority (quorum) of connected members to totally order messages. It allows members of minority components to initiate messages. These messages may diffuse through the system and become totally ordered even if their initiator is never a member of a majority component.

We suggested a replication service utility, *COReL*, implementing this algorithm, to support object replication in dynamic environments. *COReL* is constructed as a high-level communication layer over a transport layer that supplies group multicast and membership services among members of a connected network component.

We have described how *COReL* may be used in a replicated database setting, and how an *atomic commitment protocol* (ACP) may be constructed using *COReL*. *COReL* always allows members of a primary component in the network to update the database, and the ACP based on *COReL* always allows members of a primary component to resolve a transaction, regardless of past events. We know of no other ACP with this feature.

References

[1] O. Amir, Y. Amir, and D. Dolev. A Highly Available Application in the Transis Environment. In *Proceedings of the Hardware and Software Architectures for Fault Tolerance Workshop, at Le Mont Saint-Michel, France*, June 1993. LNCS 774.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms for Multicast Communication Groups. In *Intl. Workshop on Distributed Algorithms proceedings (WDAG-6), (LNCS, 647)*, number 6th, pages 292–312, November 1992.

[3] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *FTCS conference*, number 22, July 1992. previous version available as TR CS91-13, Dept. of Comp. Sci., the Hebrew University of Jerusalem.

[4] Y. Amir, D. Dolev, P. Melliar-Smith, and L. Moser. Persistent Order Maintanence in a Partitioned Network. 1994.

[5] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership using a Logical Token-Passing Ring. In *International Conference on Distributed Computing Systems*, number 13th, pages 551–560, May 1993.

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

[7] K. Birman, R. Cooper, and B. Gleeson. Programming with Process Groups: Group and Multicast Semantics. TR 91-1185, dept. of Computer Science, Cornell University, Jan 1991.

[8] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Ann. Symp. Operating Systems Principles*, number 11, pages 123–138. ACM, Nov 87.

[9] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comp. Syst.*, 9(3):272–314, 1991.

[10] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *proc. 10th annual ACM Symposium on Principles of Distributed Computing*, pages 325–340, 1991.

[11] F. Chin and K. Ramarao. Optimal Termination Protocols for Network Partitioning. In *ACM SIGACT-SIGMOD symp. on prin. of Database Systems*, pages 25–35, March 1983.

[12] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in Partitioned Networks. *ACM Computing Surveys*, 17(3):341–370, Sept. 1985.

[13] D. Dolev, D. Malki, and H. R. Strong. An Asynchronous Membership Protocol that Tolerates Partitions. submitted for publication, 1994.

[14] A. El Abbadi and N. Dani. A Dynamic Accessibility Protocol for Replicated Databases. *Data and Knowledge Engineering*, (6):319–332, 1991.

[15] A. El Abbadi and S. Toueg. Availability in Partitioned Replicated Databases. In *ACM SIGACT-SIGMOD Symp. on Prin. of Database Systems*, number 5, pages 240–251, Cambridge, MA, March 1986.

[16] D. El Abbadi, A. Skeen and F. Christian. An Efficient Fault-Tolerant Algorithm for Replicated Data Management. In *ACM SIGACT-SIGMOD symp. on prin. of Database Systems*, pages 215–229, March 1985.

[17] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32:374–382, April 1985.

[18] D. Gifford. Weighted Voting for Replicated Data. In *ACM SIGOPS Symp. on Operating Systems Principles*, December 1979.

[19] M. Herlihy. A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Trans. Comp. Syst.*, 4(1):32–53, Feb. 1986.

[20] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[21] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 78.

[22] D. Malki and R. V. Renesse. The Replication Service Layer. Private Communication.

[23] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Trans. Parallel & Distributed Syst.*, (1), Jan 1990.

[24] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended Virtual Synchrony. In *International Conference on Distributed Computing Systems*, number 14th, June 1994. To appear.

[25] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Asynchronous fault-tolerant total ordering algorithms. *to appear in SIAM Journal of Computing.*

[26] D. Peleg and A. Wool. The Availability of Quorum Systems. Technical Report CS93-17, The Weizmann Institute of Science, Rehovot, Israel, 1993.

[27] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, August 89.

[28] C. Pu and A. Leff. Replica Control in Distributed Systems: An Asynchronous Approach. In *ACM SIGMOD Symp. on Management of Data*, May 1991.

[29] F. Schneider. Implementing Fault Tolerant Services Using the State Machine Approach: A Tutorial. *Computing Surveys*, 22(4):299–319, December 1990.

[30] D. Skeen. Nonblocking Commit Protocols. In *SIGMOD Intl. Conf. Management of Data*, 1981.

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

[32] D. Skeen and M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Transactions on Software Engineering*, SE-9 NO.3, May 1983.

[33] R. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Trans. on Database Systems*, 4(2):180–209, June 1979.

[34] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. Reliable Multicast between Microkernels. In *Proceedings of the USENIX workshop on Micro-Kernels and Other Kernel Architectures*, pages 27–28, April 1992.

[35] R. van Renesse, R. Cooper, B. Glade, and P. Stephenson. A Risc Approach to Process Groups. In *Proceedings of the 5th ACM SIGOPS Workshop*, pages 21–23, September 1992.