Efficient Message Ordering in Dynamic Networks

Idit Keidar
Computer Science Institute
The Hebrew University of Jerusalem
idish@cs.huji.ac.il

Danny Dolev*
Computer Science Institute
The Hebrew University of Jerusalem
dolev@cs.huji.ac.il

Abstract

We present an algorithm for totally ordering messages in the face of network partitions and site failures. The algorithm always allows a majority of connected processors in the network to make progress (*i.e.* to 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 guarantees that when a majority is connected, each message is ordered within two communication rounds, if no failures occur during these rounds.

1 Introduction

Consistent order is a powerful paradigm for the design of fault tolerant applications, *e.g.* consistent replication [Sch90, Kei94]. We present an efficient algorithm for consistent message ordering in the face of network partitions and site failures. The network may partition into several components[1], and remerge. The algorithm is most adequate for dynamic networks where failures are transient. The algorithm uses an underlaying group communication service as a building block.

Problem Definition

Atomic broadcast deals with consistent message ordering. Informally, atomic broadcast requires that all the *correct* processors will deliver all the messages to the application in the same order and that they eventually deliver all messages sent by *correct* processors. In our model two processors may be detached, and yet both are considered correct. In this case, obviously, atomic broadcast as defined above is impossible. We solve an equivalent of atomic broadcast for partitionable networks: We require that *if a majority of the processors are alive and connected then these processors eventually deliver all messages sent by any of them*, in the same order.

The term *delivery* is usually used for delivery of totally ordered messages by the atomic broadcast algorithm to its application, but also for delivery of messages by the group communication service to its application (which in our case is the atomic broadcast algorithm). To avoid confusion, in the rest of this paper we will use the term delivery *only* for messages delivered by the group communication service to our algorithm. When discssing the atomic broadcast algorithm, we say that the algorithm *totally orders a message* when the algorithm decides that this message is the next message in the total order, instead of saying that the algorithm "delivers" the message to its application.

Failure Detection

It is well known that reaching agreement in general, and total message ordering in particular, in asynchronous environments with a possibility of even one failure is impossible [FLP85]. To overcome this difficulty, Chandra and Toueg [CT] suggest to augment the model with *failure detectors*, and prove that in this case, agreement is possible. The failure detectors suggested in [CT] notify the "correct" processors which processors are "faulty". The definition does not capture network partitions. In [FKM$^+$95] these definitions are extended for the partitionable case. The algorithm we present uses an underlying transport layer with a membership protocol that serves as the failure detector. Our algorithm guarantees that whenever the membership protocol indicates that a majority of the processors is connected, the members of this majority succeed in ordering messages. The algorithm is *correct* regardless of whether the failure detector is accurate or not, the *liveness* of the algorithm (its ability to make progress),

*This work supported by the United States - Israel Binational Science Foundation, Grant No. 92-00189

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

depends on the accuracy of this failure detector.

Algorithm Guarantees and Related Work

Our algorithm always allows processors to initiate messages, even when they are not members of a majority component. By carefully combining message ordering within a primary component and messages exchanged in minority components, messages can eventually become totally ordered even if their initiator is never a member of a majority component. The algorithm guarantees that when a majority is connected, each message is ordered within two communication rounds, if no failures occur during these rounds[2]. The algorithm incurs low overhead, no "special" messages are needed, all the information required by the protocol is piggybacked on regular messages.

Our protocol uses an underlying group communication service. Group communication mechanisms that use *hardware broadcast* lead to simpler and more efficient solutions for replication than the traditional point-to-point mechanisms. Some of the leading systems for group communication today are: ISIS [BCG91, BSS91] and its new generation HORUS [VRCGS92], Transis [ADKM92b, MADK94], Totem [AMMS$^+$93], Psync [PBS89], Newtop [EMS95] and the distributed operating system AMOEBA [KTHB89]. To increase availability, most of the group multicast algorithms mentioned above detect failures and extract faulty members from the membership. When processors reconnect, these algorithms do not recover the states of reconnected processors. This is where our algorithm comes in: it extends the order achieved by such algorithms to a global total order.

Chandra and Toueg [CT] suggest a consensus protocol that uses a failure detector, and tolerates crash failures, but does not tolerate network partitions. They suggest an atomic broadcast algorithm based on this consensus protocol. Their protocol could be extended to work in a partitionable environment [FKM$^+$95]. However, their algorithm is optimized for the crash-only model and is less efficient if partitions occur. The algorithm uses a rotating coordinator scheme among all the processors, and does not focus only on members of the current membership. When a majority is connected, the worst-case latency until ordering a single message can be $O(n)$ communication rounds.

The total ordering protocols in [MMSA93, MHS89, ADMSM94] also overcome network partitions. Total [MMSA93] incurs a high overhead: the maximum number of communication rounds required is not bounded, while our algorithm requires two communication rounds to order a message if no failures occur during these rounds. The replication algorithm suggested in [MHS89] is centralized, and thus highly increases the load on one server, while our protocol is decentralized and symmetric. The protocol in [ADMSM94] uses a majority-based scheme for message ordering, it decreases the requirement for end-to-end acknowledgments, but does not always allow a majority to make progress.

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

2 The Model

COReL (the Consistent Object Replication Layer) is an algorithm for consistent order of messages, designed to implement a high-level replication service in the Transis environment. Transis is a sophisticated transport layer (or *group communication service* layer) [ADKM92b, ADKM92a, DMS94, MADK94] that supplies omission fault free group multicast and membership services in an asynchronous environment. *COReL* uses Transis as a failure detector and as a building block for reliable communication within connected network components[3]. *COReL* may be implemented using any transport layer that supplies similar services. The layer structure of *COReL* is depicted in Figure 1.

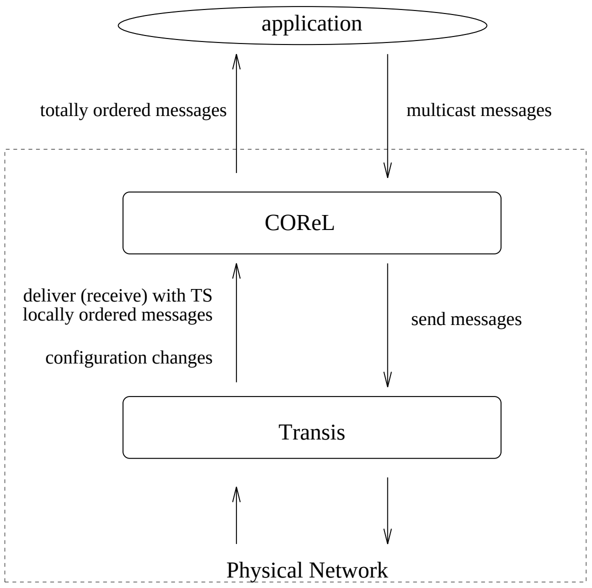

Figure 1: The Layer Structure of *COReL*

2.1 Group Multicast and Membership

Each copy of *COReL* uses the group communication service to *send* messages to the members of its group; all the members of the group *deliver* (or *receive*) the message.

After a group is created, the group undergoes *configuration (membership) changes* when processors are added

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

or are taken out of the group. The membership service reports these changes to *COReL* through special *configuration change* (*membership change*) messages. Configuration change messages are delivered among the stream of *regular* messages. Thus, during its execution, *COReL* delivers a sequence of regular messages interposed by configuration change messages. 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.

2.2 Transport Layer's Total Order Properties

This paper deals with the *ordering* of messages and events. The *causal* partial order [Lam78] is defined as the transitive closure of: $m \xrightarrow{cause} m'$ if $\text{deliver}_q(m) \rightarrow \text{send}_q(m')$ or if $\text{send}_q(m) \rightarrow \text{send}_q(m')$.

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

Property 2.1 *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 same TS is attached to the message at all the processors that deliver it. Every message has a different TS. The TS total order preserves the* causal *partial order. The transport layer delivers messages at each site in TS order.*

Property 2.2 *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.*

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 2.1 and 2.2, they will not both be members of the same next configuration). In Property 2.3 we require that if q delivers m' without m, then no message m'' sent by q, after delivering m', can be delivered by p in the context of C_1.

Property 2.3 *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'.*

A framework for a partitionable membership service that fulfills these properties is described in [DMS95]. These properties are also fulfilled in the Agreed Communication service of the *Extended Virtual Synchrony (EVS) model* [MAMSA94]. The Transis [ADKM92b, MADK94] and Totem [AMMS+93] systems implement partitionable membership and ordering services of this framework, and also support EVS.

3 The *COReL* Algorithm

We present the *COReL* (Consistent Object Replication Layer) algorithm for reliable multicast and total ordering of messages. The *COReL* algorithm is used to implement long-term replication services using a transport layer that supplies total order of messages within each connected network component. *COReL* guarantees that all messages will reach all sites in the same order. It always allows members of a connected primary component to order messages. The algorithm is resilient to both processor failures and network partitions.

3.1 Reliable Multicast

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. 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. Note that it is important to use application level ACKs in order to guarantee that the message is logged on stable storage. If the message is only ACKed at the transport layer level, it may be lost if the process crashes.

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. They determine which messages should be retransmitted and by whom.

3.2 Message Ordering

Within each component messages are ordered by the transport layer. The transport layer supplies a unique timestamp (*TS*) for each message when it delivers the message to *COReL*. When *COReL* receives the message, it writes the message on stable storage along with its TS. Within a majority component *COReL* orders messages according to their TS. The TS is globally unique, even in the face of partitions, and yet *COReL* sometimes orders messages in a different total order:

it orders messages from majority component before (causally concurrent) messages with a possibly higher TS from minority components (otherwise it wouldn't always allow a majority to make progress). Note that both the TS order and the order provided by *COReL* preserve the causal partial order.

When a message is retransmitted, the TS 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 (the new TS generated by the transport layer during retransmission is ignored).

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 component bit* is set iff this processor is currently a member of a primary component. In Section 3.5.1 we describe how a majority of the processors may become a primary component. 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.*

If a message is totally ordered 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 *yellow message mechanism* explained below.

3.2.1 The Colors Model

COReL 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 is the *message id.*

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 messages that were ordered already).

COReL builds its knowledge about the order of messages at other processors. We use the colors model defined in [AAD93] to indicate the knowledge level associated with each message, as follows:

green: Knowledge about the message's global total order. 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. Note that this is when the message is totally ordered according to Order Rule 1. 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.*

yellow: 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. The yellow messages are the next candidates to become green. The last yellow message in $\mathcal{MQ}$ marks the *yellow line.*

red: No knowledge about the message's global total order. A message in $\mathcal{MQ}$ is *red* if there is no knowledge that it has a different color. Yellow messages precede all the red messages in $\mathcal{MQ}$. Thus, $\mathcal{MQ}$ is divided into three zones: a green prefix, then a yellow zone and a red suffix.

When a message is marked as green it is totally ordered.

If a member of a primary component PM marks a message m as green according to Order Rule 1 then for all the other members of PM, m is yellow or green. Since two majorities always intersect, and every primary component contains a majority, in the next primary component that will be formed at least one member will have m as yellow or green. 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.

3.3 Notation

We use the following notation:

- $\mathcal{MQ}^p$ is the $\mathcal{MQ}$ of processor p.
- $Prefix(\mathcal{MQ}^p, m)$ is the prefix of $\mathcal{MQ}^p$ ending at message m.
- $Green(\mathcal{MQ}^p)$ 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[4], then p *knows* of PM.

[4] p recovers the state of q when p completes running the Recovery Procedure for a configuration that contains q.

3.4 Invariants of the Algorithm

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* maintained by each step of the algorithm:

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 sent before m, $\mathcal{MQ}^p$ contains m' before m.
- 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 $Green(\mathcal{MQ}^p)$, and this is the only way that $Green(\mathcal{MQ}^p)$ may change.

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

In [Kei94] we formally prove that these invariants hold in *COReL*, and thus prove the correctness of *COReL*.

3.5 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 3.4.

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.

Configuration Change Handler for 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 3.5.2.
- If C is a majority, run the algorithm for establishing a new primary component, described in Section 3.5.1.
- Continue handling and sending regular messages.

Figure 2: Configuration Change Handler

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 ends, and thus the **Causal** invariant is 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, and 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 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. A detailed description of the Recovery Procedure is presented in Section 3.5.2.

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 the new configuration is a majority, the members of C will try to establish a new primary component. The algorithm for establishing a new primary component is described in Section 3.5.1. All committed primary components are sequentially numbered. We refer to the primary component with sequential number i as PM_i.

When a configuration change is delivered, the handler described in Figure 2 is invoked. In the course of the run of the handler, the primary component bit is unset, regular messages are blocked, and no new regular messages are initiated.

3.5.1 Establishing a Primary Component

A new configuration, C, is established as the new primary component, if C is a majority, after the retransmission phase described in Section 3.5.2. The primary component is established in a three-phase agreement protocol, similar to Three Phase Commit protocols [Ske81, KD95]. The three phases are required in order to allow for recovery in case failures occur in the course of the establishing process. The three phases correlate to the three levels of colors in $\mathcal{MQ}$.

In the first phase all the processors multicast a message to notify the other members that they **attempt** 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 and set the primary component bit to TRUE. 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. If the transport layer reports of a configuration change before the process is over – the establishing is aborted, but none of its effects need to be 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 3.

3.5.2 Recovery Procedure

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

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

- Compute: $New\_Primary = \max_{i \in C}(Last\_Attempted\_Primary_i) + 1$.
  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 3: Establishing a new primary component

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[5].
   - 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.
3. Compute: $Max\_Committed = \max_{p \in C} Last\_Committed\_Primary_p$.

   Let *Representatives* be the members that have: $Last\_Committed\_Primary = Max\_Committed$.

[5] 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.

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 4: Retransmission Rule

The *Representatives* advance their green lines to include all messages that any member of C had marked as green, and retreat their yellow lines to include only messages that all of them had marked as yellow, and in the same order, *e.g.* 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. If all the members have the same last committed primary component, (*i.e.* all are *Representatives*), go directly to Step 7.

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

   Determine (from the state messages) the following sets of messages:

   component_stable is the set of messages that all the members of C have.

   component_ordered is the set of 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 component_ordered messages only. 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[6].

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 $Last\_Committed\_Primary$ to $Max\_Committed$ (on stable storage).
   - 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, are retransmitted. Each message is retransmitted by at most one processor. The processors that need to retransmit messages send them, with their original ACKs, in an order maintaining the Retransmission Rule (described in Figure 4).

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

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.

After receiving all of the retransmitted messages: if C is a majority then the members try to establish a new configuration. (The algorithm is described Section 3.5.1).

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 2.2 of the transport layer, all the members received the same set of messages until the configuration change.

4 Discussion

We presented an efficient algorithm for totally ordered multicast in an asynchronous environment, that is resilient to network partitions and communication link failures. The algorithm always allows a majority of connected members to totally order messages within two communication rounds. The algorithm is constructed over a transport layer that supplies group multicast and membership services among members of a connected network component.

The algorithm allows members of minority components to initiate messages. These messages may diffuse

[6]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.

through the system and become totally ordered even if their initiator is never a member of a majority component: The message is initially multicast in the context of the minority component, if some member of the minority component (not necessarily the message initiator) later becomes a member of a majority component, the message is retransmitted in the majority component and becomes totally ordered.

Some of the principles presented in this protocol may be applied to make a variety of distributed algorithms more available, *e.g.* network management services and distributed database systems. In [KD95] we present an atomic commitment protocol for distributed database management based on such principles.

In [Kei94] we suggest two extensions of the algorithm, optimizing it for highly unreliable networks, where a majority of the processors are rarely connected at once. The first extension deals with the case that a majority component is not formed for a long period although all the processors are alive, and messages from all of them eventually reach all the destinations. In this case, we enable the processors to totally order messages according to an alternative mechanism, that does not require a primary component to be established. This extension will be further developed in future work.

The second extension deals with *dynamic voting*. The algorithm presented in Section 3 uses a majority to decide if a group of processors may become a primary component. The concept of majority can be generalized to quorums, and can be further generalized, to allow more flexibility yet: The dynamic voting paradigm for electing a primary component defines quorums adaptively. When a partition occurs, a majority of the previous quorum may chosen as the new primary component. Thus, a primary component must not necessarily a majority of the processors. Dynamic voting may introduce inconsistencies, and therefore should be handled carefully. In [DKYL] we suggest an algorithm for consistently maintaining a primary component using dynamic voting. This algorithm may be easily incorporated into *COReL*.

Finally, in [Kei94] we prove the correctness of the *COReL* algorithm.

Acknowledgments

Yair Amir greatly contributed to this work with his original ideas and insight. Special thanks to Dalia Malki and Catriel Beeri who suggested simplifications to the algorithm and presentation.

References

[AAD93] 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.

[ADKM92a] 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.

[ADKM92b] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *FTCS conference*, number 22, July 1992.

[ADMSM94] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and Efficient Replication using Group Communication. Technical Report CS94-20, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[AMMS$^+$93] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership using a Logical Token-Passing Ring. In *International Conference on Distributed Computing Systems*, number 13th, pages 551–560, May 1993.

[BCG91] 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.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comp. Syst.*, 9(3):272–314, 1991.

[CT] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. *Journal of ACM.* To appear. Previous version: PODC 1991 pp. 325-340.

[DKYL] Danny Dolev, Idit Keidar, and Esti Yeger-Lotem. Dynamic Voting for Consistent Primary Components. In preparation.

[DMS94] D. Dolev, D. Malki, and H. R. Strong. An Asynchronous Membership Protocol that Tolerates Partitions. Technical Report CS94-6, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[DMS95] D. Dolev, D. Malki, and H. R. Strong. A Framework for Partitionable Membership Service. TR 95-4, Institute of Computer Science, The Hebrew University of Jerusalem, March 1995.

[EMS95] P. D. Ezhilchelvan, A. Macedo, and S. K. Shrivastava. Newtop: a fault tolerant group communication protocol. In *International Conference on Distributed Computing Systems*, number 15th, June 1995.

[FKM$^+$95] R. Friedman, I. Keidar, D. Malki, K. Birman, and D. Dolev. Deciding in Partitionable Networks. TR 95-16, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, November 1995. Also Cornell TR95-1554. Available via anonymous ftp at cs.huji.ac.il (132.65.16.10) in users/transis/TR95-16.ps.gz.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32:374–382, April 1985.

[KD95] I. Keidar and D. Dolev. Increasing the Resilience of Atomic Commit, at No Additional Cost. In *ACM Symp. on Prin. of Database Systems (PODS)*, pages 245–254, May 1995. Previous version available as Technical Report CS94-18, The Hebrew University, Jerusalem, Isreal.

[Kei94] I. Keidar. A Highly Available Paradigm for Consistent Object Replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994. Also available as Technical Report CS95-5, and via anonymous ftp at cs.huji.ac.il (132.65.16.10) in users/transis/thesis/keidar-msc.ps.gz.

[KTHB89] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and E. H. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 78.

[MADK94] D. Malki, Y. Amir, D. Dolev, and S. Kramer. The Transis Approach to High Availability Cluster Communication. TR CS94-14, Institute of Computer Science, The Hebrew University of Jerusalem, June 1994.

[MAMSA94] 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.

[MHS89] Tim Mann, Andy Hisgen, and Garret Swart. An Algorithm for Data Replication. Technical Report 46, DEC Systems Research Center, June 1989.

[MMSA93] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Asynchronous Fault-Tolerant Total Ordering Algorithms. *SIAM Journal of Computing*, 22(4):727–750, August 1993.

[PBS89] 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.

[Sch90] F. B. Schneider. Implementing Fault Tolerant Services Using The State Machine Approach: A Tutorial. *Computing Surveys*, 22(4):299–319, December 1990.

[Ske81] D. Skeen. Nonblocking Commit Protocols. In *SIGMOD Intl. Conf. Management of Data*, 1981.

[VRCGS92] 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.