Chapter 3

TOTALLY ORDERED BROADCAST IN THE FACE OF NETWORK PARTITIONS

Exploiting Group Communication for Replication in Partitionable Networks[1]

Idit Keidar
Laboratory for Computer Science
Massachusetts Institute of Technology
idish@theory.lcs.mit.edu

Danny Dolev
Computer Science Department
Hebrew University of Jerusalem
dolev@cs.huji.ac.il

Abstract We present an algorithm for Totally Ordered Broadcast in the face of network partitions and process failures, using an underlying group communication service as a building block. The algorithm always allows a majority (or quorum) of connected processes in the network to make progress (i.e., to order messages), if they remain connected for sufficiently long, regardless of past failures. Furthermore, the algorithm always allows processes to initiate messages, even when they are not members of a majority component in the network. These messages are disseminated to other processes using a gossip mechanism. 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 at most two communication rounds, if no failures occur during these rounds.

Keywords: Group communication, totally ordered broadcast, replication, network partitions.

1. INTRODUCTION

Totally Ordered Broadcast is a powerful service for the design of fault tolerant applications, e.g., consistent cache, distributed shared memory and replication, as explained in Schneider, 1990; Keidar, 1994. We present the *COReL (Consistent Object Replication Layer)* algorithm for Totally Ordered Broadcast in the face of network partitions and process failures. The algorithm is most adequate for dynamic networks where failures are transient.

COReL uses an underlying totally ordered *group communication service (GCS)*, cf. acm, 1996 as a building block. Group communication introduces the notion of group abstraction which allows processes to be easily arranged into multicast groups. Within each group, the GCS provides reliable multicast and membership services. The task of the membership service is to maintain the set of currently live and connected processes in each group and to deliver this information to the group members whenever it changes. The reliable multicast services deliver messages to all the current members of the group. GCSs (e.g., Transis - Dolev and Malkhi, 1996; Amir et al., 1992, Ensemble - Hayden and van Renesse, 1996, Horus - van Renesse et al., 1996 and Totem - Amir et al., 1995; Moser et al., 1996) that use hardware broadcast where possible lead to simpler and more efficient solutions for replication than the traditional point-to-point mechanisms.

COReL multicasts messages to all the connected members using the underlying GCS. Once messages are delivered by the GCS and logged on stable storage (by *COReL*), they are acknowledged. Acknowledgments are piggybacked on regular messages. When a majority is connected, messages become totally ordered once they are acknowledged by all the members of the connected majority. Thus, the *COReL* algorithm guarantees that when a majority is connected, each message is ordered within two communication rounds at the most, if no failures occur during these rounds[2]. The algorithm incurs low overhead: No “special” messages are needed and all the information required by the protocol is piggybacked on regular messages.

Processes using *COReL* are always allowed to initiate messages, even when they are not members of a majority component. By carefully combining message ordering within a primary component and gossiping of messages exchanged in minority components, messages can eventually become totally ordered even if their initiator is never a member of a majority component.

The protocol presented herein uses a simple majority rule to decide which network component can become the primary one. Alternatively, one could use a quorum system (cf. Peleg and Wool, 1995), which is a generalization of the majority concept. A quorum system is a collection of sets (quorums) such that any two sets intersect. Using such a quorum system, a network component can become the primary one if it contains a quorum. The concept of quorums may be further generalized to allow dynamic adjustment of the quorum system. In Yeger Lotem et al., 1997, we present a dynamic voting protocol for maintaining the primary component in the system; this protocol may be used in conjunction with *COReL*.

1.1 THE PROBLEM

The *Atomic Broadcast* problem defined in Hadzilacos and Toueg, 1993 deals with consistent message ordering. Informally, Atomic Broadcast requires that all the correct processes will deliver all the messages to the application in the same order and that they eventually deliver all messages sent by correct processes. Furthermore, all the correct processes must deliver any message that is delivered by a correct processes.

In our model two processes may be detached, and yet both are considered correct. In this case, obviously, Atomic Broadcast as defined above is unsolvable (even if the communication is synchronous, please see Friedman et al., 1995). We define a variant of Atomic Broadcast for partitionable networks: We guarantee that if a majority of the processes form a connected component then these processes eventually deliver all messages sent by any of them, in the same order. We call this service *Totally Ordered Broadcast*.

It is well-known that in a fully asynchronous failure-prone environment, agreement problems such as Consensus and Atomic Broadcast are not solvable (as proven in Fischer et al., 1985), and it is also impossible to implement an algorithm with the above guarantee (please see Friedman et al., 1995). Such agreement problems are solvable, on the other hand, if the model is augmented with an external *failure detector* (please see Chandra and Toueg, 1996; Babaoğlu et al., 1995; Dolev et al., 1996; Friedman et al., 1995; Dolev et al., 1997).

The algorithm we present herein uses an underlying group communication service with a membership protocol that serves as the failure detector. Our algorithm guarantees that whenever there is a connected component which contains a majority of the processes, and the membership protocol indicates that this component is connected, the members of this majority succeed in ordering messages. The *safety* properties of *COReL* are preserved regardless of whether the failure detector is accurate or not; the *liveness* of the algorithm (its ability to make progress) depends on the accuracy of this membership protocol.

Informally, *COReL* satisfies the following conditional liveness property: If in a given run of *COReL* there is a time after which the network stabilizes with a connected majority component and the membership is accurate, then *COReL* eventually totally orders every message sent in the majority component. This guarantee is formally stated in Property 3.8. Here, we do not analyze how long it takes before *COReL* totally orders a message. Such an analysis may be found in Fekete et al., 1997.

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 GCS to its application (which in our case is the Totally Ordered Broadcast algorithm). To avoid confusion, henceforward we will use the term delivery only for messages delivered by the GCS to our algorithm. When discussing the Totally Ordered 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.

1.2 RELATED WORK

Group communication systems often provide totally ordered group communication services. Amoeba (Kaashoek and Tanenbaum, 1996), Delta-4 (Powell, 1991) Ensemble (Hayden and van Renesse, 1996), Horus (van Renesse et al., 1996), Isis (Birman et al., 1991), Totem (Amir et al., 1995; Moser et al., 1996), Transis (Dolev and Malkhi, 1996; Amir et al., 1992) and RMP (Whetten et al., 1995) are only some examples of systems that support totally ordered group communication.

To increase availability, GCSs detect failures and extract faulty members from the membership. When processes reconnect, the GCS does not recover the states of reconnected processes. This is where the *COReL* algorithm comes in: *COReL* recovers lost messages and extends the order achieved by the GCS to a global total order.

The majority-based Consensus algorithms of Dwork et al., 1988; Lamport, 1989; De Prisco et al., 1997; Chandra and Toueg, 1996; Dolev et al., 1996 are guaranteed to terminate under conditions similar to those of *COReL*, i.e., at periods at which the network is stable and message delivery is timely, or when failure detectors are eventually accurate. Atomic Broadcast is equivalent to Consensus (as proven in Chandra and Toueg, 1996); Atomic Broadcast may be solved by running a sequence of Consensus decisions (as done, e.g., in Chandra and Toueg, 1996; Lamport, 1989; De Prisco et al., 1997).

The main advantage of using *COReL* over running a sequence of Consensus algorithms is that *COReL* essentially pipelines the sequence of Consensus decisions. While Consensus algorithms involve special rounds of communication dedicated to exchanging "voting" messages of the protocol, in our approach all the information needed for the protocol is piggybacked on regular messages. Furthermore, *COReL* does not maintain the state of every Consensus invocation separately, the information about all the pending messages is summarized in common data structures. This allows faster recovery from partitions, when *COReL* reaches agreement on all the recovered messages simultaneously.

The Atomic Broadcast algorithm of Chandra and Toueg, 1996 conserves special "voting" messages by reaching agreement on the order of sets of messages instead of running Consensus for every single message. However, this increases the latency of message ordering and still requires some extra messages.

In Mann et al., 1989, the Paxos multiple Consensus algorithm of Lamport, 1989 is used for a replicated file system. The replication algorithm suggested in Mann et al., 1989 is centralized, and thus highly increases the load on one server, while *COReL* is decentralized and symmetric.

The total ordering protocol in Amir, 1995; Amir et al., 1994 resembles *COReL*; it also exploits a group communication service to overcome network partitions. Like *COReL*, it uses a majority-based scheme for message ordering. It decreases the requirement for end-to-end acknowledgments, at the price of not always allowing a majority to make progress.

Fekete et al., 1997 have studied the *COReL* algorithm (following its publication in Keidar and Dolev, 1996) using the I/O automata formalism. They have presented both the specifications and the implementation using I/O automata. They have analyzed the algorithm's liveness guarantees in terms of timed automata behavior at periods during which the underlying network is stable and timely. They made simplifications to the protocol which make it simpler to present, alas less efficient.

The Total protocol (Moser et al., 1993) also totally orders messages in the face of process crashes and network partitions. However, it 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.

2. THE MODEL

The underlying communication network provides datagram message delivery. There is no known bound on message transmission time, hence the system is asynchronous. Processes fail by crashing, and crashed processes may later recover. Live processes are considered *correct*, crashed processes are *faulty*. Recovered processes come up with their stable storage intact. Communication links may fail and recover. Malicious failures are not considered; messages are neither corrupted nor spontaneously generated by the network, as stated in the following property:

Property 3.1 (Message Integrity) *If a message m is delivered by a process p, then there is a causally preceding send event of m at some process q.*

The *causal* partial order (first defined in Lamport, 78) is defined as the transitive closure of: $m \xrightarrow{cause} m'$ if $\text{deliver}_q(m) \rightarrow \text{send}_q(m')$ or if $\text{send}_q(m) \rightarrow \text{send}_q(m')$.

3. THE SYSTEM ARCHITECTURE

COReL is an algorithm for Totally Ordered Broadcast. *COReL* is designed as a high-level service atop a group communication service which provides totally ordered group multicast and membership services, and is omission fault free within connected network components.

COReL uses the GCS as a failure detector and as a building block for reliable communication within connected network components. The layer structure of *COReL* is depicted in Figure 3.1.

All the copies of *COReL* are members of one multicast group. Each copy of *COReL* uses the GCS 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 view changes when processes are added or are taken out of the group due to failures. The membership service reports these changes to *COReL* through special view messages. A view v is a pair consisting of a view identifier $v.id$ and a set of members $v.set$. We say that a process p is a member of a view v if $p \in v.set$.

Views are delivered among the stream of regular messages. We say that a send (receive) event e occurs at process p in view v (or in the context of v) if v was the latest view that p received before e.

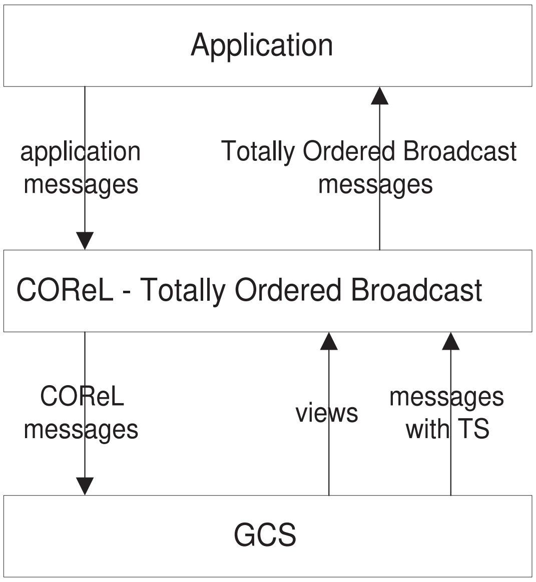

Figure 3.1 The layer structure of *COReL*.

3.1 PROPERTIES OF THE GCS

COReL may be implemented using any GCS that provides reliable locally ordered group multicast and membership services. We assume the GCS fulfills the following properties:

Messages are not duplicated in transit:

Property 3.2 (No Duplication) *Every message delivered at a process p is delivered only once at p.*

Messages are totally ordered within each connected network component – if two processes deliver the same two messages then they deliver them in the same order. This feature is guaranteed using logical *timestamps (TSs)* which are delivered along with the messages:

Property 3.3 (Total Order) *A logical* timestamp (TS) *is attached to every message when it is delivered. Every message has a unique TS, which is attached to it at all the processes that deliver it. The TS total order preserves the causal partial order. The GCS delivers messages at each process in the TS order (possibly with gaps).*

The following property is perhaps the most well known property of GCSs, to the extent that it engendered the whole Virtual Synchrony (cf. Birman and van Renesse, 1994; Birman and Joseph, 1987; Friedman and van Renesse, 1995; Moser et al., 1994) model:

Property 3.4 (Virtual Synchrony) *Any two processes undergoing the same two consecutive views in a group G deliver the same set of messages in G within the former view.*

Virtual Synchrony guarantees that process that remain connected agree upon the set of messages they deliver. Among processes that do not remain connected we would also like to guarantee agreement to some extent. If two processes become disconnected, we do not expect to achieve full agreement on the set of messages they delivered in the context of the old view before detaching. Instead, we require that they agree on a prefix of the messages that they deliver in this view, as described below.

Let processes p and q be members of view v_1. Assume that p delivers a message m before m' in v_1, and that q delivers m', but without delivering m. This can happen only if p and q became disconnected (from Properties 3.3 and 3.4, they will not both be members of the same next view). In Property 3.5 below, 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 v_1, as illustrated in Figure 3.2.

Property 3.5 *Let p and q be members of view v. If p delivers a message m before m' in v, and if q delivers m' and later sends a message m'', such that p delivers m'' in v, then q delivers m before m'.*

The GCS also preserves the Message Integrity property (Property 3.1) of the underlying communication.

These properties are fulfilled by several GCSs, e.g., Totem (Amir et al., 1995; Moser et al., 1996), the ATOP (Chockler et al., 1998; Chockler, 1997) and All-Ack (Dolev and Malki, 1995; Malki, 1994) total order protocols in Transis (Dolev and Malkhi, 1996), the strong total order implemented in Phoenix (Malloth et al., 1995), and two of the total order protocols in Horus (Friedman and van Renesse, 1997).

4. PROBLEM DEFINITION: THE SERVICE GUARANTEES

Safety

COReL fulfills the following two safety properties:

Property 3.6 *At each process, messages become totally ordered in an order which is a prefix of some common global total order. I.e., for*

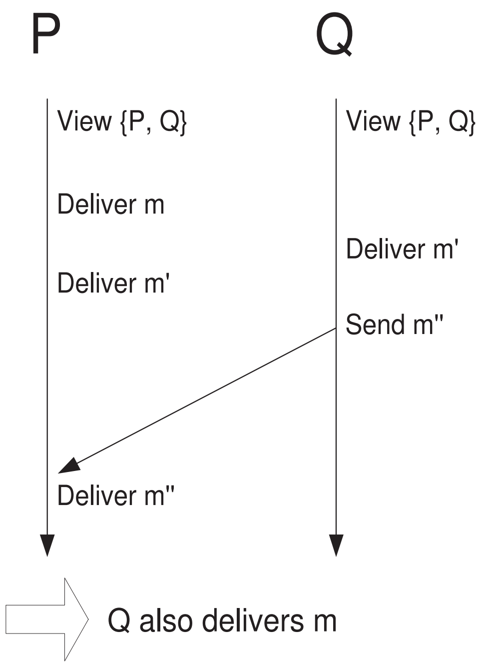

Figure 3.2 Property 3.5.

any two processes p and q, and at any point during the execution of the protocol, the sequence of messages totally ordered by p is a prefix of the sequence of messages totally ordered by q, or vice versa.

Property 3.7 *Messages are totally ordered by each process in an order which preserves the causal partial order.*

In addition, *COReL* preserves the following properties of underlying GCS: Message Integrity (Property 3.1) and No Duplication (Property 3.2).

Liveness

Property 3.8 (Liveness) *Let $\mathcal{P}$ be a set of processes and v a view s.t. $v.set = \mathcal{P}$. Assume there is a time t such that no member of $\mathcal{P}$ delivers any view after time t and the last view delivered by each $p \in \mathcal{P}$ before time t is v. Furthermore, assume that every message sent by a process $p \in \mathcal{P}$ in view v is delivered by every process $q \in \mathcal{P}$. Then, COReL guarantees that every message sent by a process in $\mathcal{P}$ in any view is eventually totally ordered by all the members of $\mathcal{P}$.*

5. THE COREL ALGORITHM

We present the *COReL* algorithm for reliable multicast and total ordering of messages. The *COReL* algorithm is used to implement long-term replication services using a GCS as a building block. *COReL* guarantees that all messages will reach all processes in the same order. It always allows members of a connected primary component to order messages. The algorithm is resilient to both process failures and network partitions.

5.1 RELIABLE MULTICAST

When the network partitions, messages are disseminated in the restricted context of a smaller view, and are not received at processes which are members of other components. The participating processes keep these messages for as long as they might be needed for retransmission. Each process logs (on stable storage) every message that it receives from the GCS. A process 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 GCS level, it may be lost if the process crashes.

When network failures are mended and previously disconnected network components re-merge, a Recovery Procedure is invoked; the members of the new view exchange messages containing information about messages in previous components and their order. They determine which messages should be retransmitted and by whom.

When a process crashes, a message that it sent prior to crashing may be lost. When a process recovers from such a crash, it needs to recover such messages. Therefore, messages are stored (on stable storage) when they are received by the application (before the application send event is complete).

5.2 MESSAGE ORDERING

Within each component messages are ordered by the GCS layer, which 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: *COReL* orders messages from majority component before (causally concurrent) messages with a possibly higher TS from minority components. This is necessary in order to 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 GCS 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 process, the *primary component bit* indicates if this process is currently a member of a primary component. In Section 5.5.1 we describe how a majority of the processes may become a primary component. Messages that are received in the context of a primary component (i.e., when the primary component bit is TRUE) 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 process 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 in Section 5.2.1 below.

Every instance of *COReL* maintains a local message queue $\mathcal{MQ}$ that is an ordered list of all the messages that this process received from the application and the GCS. After message m was received by *COReL* at process 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 $\langle sender, counter \rangle$. 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. Messages are also inserted to the $\mathcal{MQ}$ (without a TS) when they are received from the application. Once Self Delivery occurs, these messages are tagged with the TS provided by the GCS and are moved to their proper place in the $\mathcal{MQ}$. 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).

5.2.1 The Colors Model.

COReL builds its knowledge about the order of messages at other processes. We use the colors model defined in Amir et al., 1993 to indicate the knowledge level associated with each message, as follows:

green: Knowledge about the message's global total order. A process 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 occurs exactly when the message is totally ordered according to Order Rule 1. The set of green messages at each process at a given time is a prefix of $\mathcal{MQ}$. The last green message in $\mathcal{MQ}$ marks the *green line.*

yellow: Each process 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.

An example snapshot of different message queues at different processes is shown in Figure 3.3. In this example, P and Q form a majority component. R is a member of a minority component. Messages 1 and 2 have become green in a former majority component that all processes have knowledge of. Messages 3 and 4 have become green at P in the current majority component, therefore, they are either green or yellow at Q. P has messages 5 and 6 as yellow, which implies that it does not know whether Q has these messages or not. Message x was sent in a minority component, and therefore it is red.

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, processes recover missing messages and have to agree upon their order; 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 different

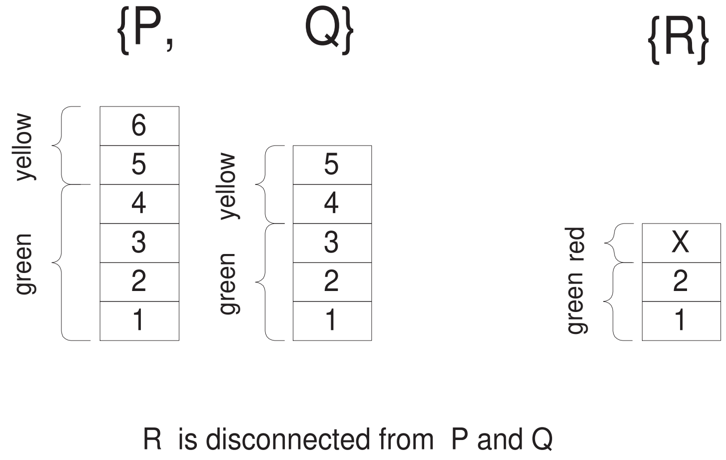

Figure 3.3 The $\mathcal{MQ}$s at three processes running *COReL*.

components are interleaved according to the TS order. After recovery is complete, all the messages in $\mathcal{MQ}$ are marked as green.

Consider, for example, the state illustrated in Figure 3.3 above. Assume that at this point in the execution, P partitions from Q and forms the singleton minority component $\{P\}$, while Q re-connects with R to form the majority component $\{Q, R\}$. Figure 3.4 depicts the state of the $\mathcal{MQ}$s of the members of the two components once recovery is complete.

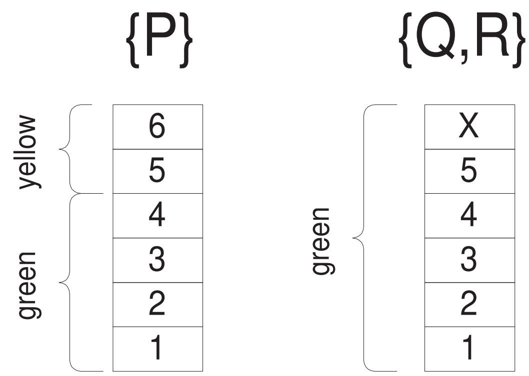

Figure 3.4 The $\mathcal{MQ}$s after after recovery.

As explained in Amir et al., 1993; Keidar, 1994, it is possible to provide the application with red messages if weak consistency guarantees are required. For example, eventually serializable data services (e.g., Pu and Leff, 1991; Fekete et al., 1996; Amir et al., 1993) deliver messages to the application before they are totally ordered. Later, the application is notified when the message becomes *stable* (green in our terminology). Messages become stable at the same order at all processes. The advantage of using *COReL* for such applications is that with *COReL* messages become stable even whenever a majority is connected, while with the implementations presented in Pu and Leff, 1991; Fekete et al., 1996; Amir et al., 1993, messages may become stable only after they are received by all the processes in the system.

5.3 NOTATION

We use the following notation:

- $\mathcal{MQ}_p$ is the $\mathcal{MQ}$ of process 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 *process p knows of a primary component PM* recursively as follows:
  1. If a process p was a member of PM then p *knows* of PM.
  2. If a process q *knows* of PM, and p recovers the state of q[3], then p *knows* of PM.

5.4 INVARIANTS OF THE ALGORITHM

The order of messages in $\mathcal{MQ}$ of each process 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 process, the order of green messages in $\mathcal{MQ}$ is never altered. Furthermore, the algorithm totally orders messages in the same order at all processes, therefore the different processes must agree on their green prefixes.

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

Causal
- If a process p has in its $\mathcal{MQ}$ a message m that was originally sent by process q, then for every message m' that q sent before m, $\mathcal{MQ}_p$ contains m' before m.
- If a process p has in its $\mathcal{MQ}$ a message m that was originally sent by process 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 processes have compatible green prefixes: Let p and q be a pair of processes running the algorithm. At any point in the course of the execution – one of $Green(\mathcal{MQ}_p)$ and $Green(\mathcal{MQ}_q)$ is a prefix of the other.

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

1. Process q has m marked as yellow or green.
2. $Prefix(\mathcal{MQ}_q, m) = Prefix(\mathcal{MQ}_p, m)$.

In Keidar, 1994 we formally prove that these invariants hold in *COReL*, and thus prove the correctness of *COReL*.

5.5 HANDLING VIEW CHANGES

The main subtleties of the algorithm are in handling view 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 view changes must maintain the invariants described in Section 5.4.

When a view change is delivered, the handler described in Figure 3.5 is invoked. In the course of the run of the handler, the primary component bit is FALSE, regular messages are blocked, and no new regular messages are initiated.

View Change Handler for View v:

- Set the primary component bit to FALSE.
- Stop handling regular messages, and stop sending regular messages.
- If v contains new members, run the Recovery Procedure described in Section 5.5.2.
- If v is a majority, run the algorithm to establish a new primary component, described in Section 5.5.1.
- Continue handling and sending regular messages.

Figure 3.5 View change handler.

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

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

In the course of the Recovery Procedure, the members agree upon common 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 v 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 5.5.2.

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. This is necessary in order to be consistent with decisions made in previous primary components.

If the new view is a majority, the members of v will try to establish a new primary component. The algorithm for establishing a new primary component is described in Section 5.5.1. All committed primary components are sequentially numbered. We refer to the primary component with sequential number i as PM_i.

5.5.1 Establishing a Primary Component. A new view, v, is established as the new primary component, if v is a majority, after the retransmission phase described in Section 5.5.2. The primary component is established in a three-phase agreement protocol, similar to Three Phase Commit protocols (cf. Skeen, 1982; Keidar and Dolev, 1998). 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 $\mathcal{MQ}$.

Establishing a New Primary Component in view v
If v contains new members, the Recovery Procedure is run first.
Let $New\_Primary = \max_{i \in v.set}(Last\_Attempted\_Primary_i) + 1$.
If v is a majority, all members of a view v try to establish it as the new primary component $PM_{New\_Primary}$:

Phase 1 – Attempt (red):
Set *Last_Attempted_Primary* to *New_Primary* on stable storage, and send an attempt message to the other members of v. Wait for attempt messages from all members of v.

Phase 2 – Commit (yellow):
Once attempt messages from all members of v arrive, commit to the view by setting *Last_Committed_Primary* to *New_Primary* on stable storage and marking all the messages in the $\mathcal{MQ}$ that are not green as yellow.
Send a commit message to the other members of v.

Phase 3 – Establish (green):
Once commit messages from all members of v arrive, *establish* v, by setting to TRUE the primary component bit and marking as green all the messages in $\mathcal{MQ}$.

If the GCS reports of a view change before the process is over – the establishing is aborted, but its effects are not undone.

Figure 3.6 Establishing a new primary component.

In the first phase all the processes 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 processes mark all the messages in their $\mathcal{MQ}$ as green and set the primary component bit to TRUE. A process 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 GCS reports of a view change before the process is over – the establishing is aborted, but none of its effects need to be undone. The primary component bit remains FALSE until the next successful establish process.

Each process maintains the following variables:

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

Last_Attempted_Primary is the number of the last primary component that this process has attempted to establish. This number may be higher than the number of the last component actually committed to.

The algorithm for establishing a new primary component is described in Figure 3.6.

5.5.2 Recovery Procedure. If the new view, v, introduces new members, then each process that delivers the view change runs the following protocol:

Recovery Procedure for process p and view v

1. Send state message including the following information:
   - $Last\_Committed\_Primary$.
   - $Last\_Attempted\_Primary$.
   - For every process q, the id of the last message that p received from q[4].
   - 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 processes in $v.set$.
3. Let: $Max\_Committed = \max_{p \in v.set} Last\_Committed\_Primary_p$.

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

   The *Representatives* advance their green lines to include all messages that any member of v 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. For example, if process 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 m 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 v have.

   component_ordered is the set of messages that are green for all the members of v.

   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 view 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[5].

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 process. The processes that need to retransmit messages send them, with their original

ACKs, in an order maintaining the Retransmission Rule described in Figure 3.7.

Retransmission Rule *If process 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 process q has to retransmit m then p does not retransmit m' before receiving the retransmission of m.*

Figure 3.7 Retransmission rule.

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

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

After receiving all of the retransmitted messages, if v is a majority then the members try to establish a new view. (The algorithm is described Section 5.5.1).

If the view change reports only of process faults, and no new members are introduced, the processes need only establish the new view and no retransmissions are needed. This is due to the fact that, from Property 3.4 of the GCS, all the members received the same set of messages until the view change.

6. 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 GCS 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 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 Keidar and Dolev, 1998 we present an atomic commitment protocol for distributed database management based on such principles.

The algorithm presented herein 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 Yeger Lotem et al., 1997 we suggest an algorithm for consistently maintaining a primary component using dynamic voting. This algorithm may be easily incorporated into *COReL*, optimizing it for highly unreliable networks.

In Keidar, 1994 we prove the correctness of the *COReL* algorithm.

Acknowledgments

The authors are thankful to Yair Amir Dalia Malki and Catriel Beeri for many interesting discussions and helpful suggestions.

Notes

1. This chapter is based on the paper Keidar and Dolev, 1996

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

3. p recovers the state of q when p completes running the Recovery Procedure for a view that contains q.

4. 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.

5. 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.

References

(1996). *Communications of the ACM 39(4), special issue on Group Communications Systems.* ACM.

Amir, O., Amir, Y., and Dolev, D. (1993). 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.* LNCS 774.

Amir, Y. (1995). *Replication Using Group Communication Over a Partitioned Network.* PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel.

Amir, Y., Dolev, D., Kramer, S., and Malki, D. (1992). Transis: A communication sub-system for high availability. In *22nd IEEE Fault-Tolerant Computing Symposium (FTCS).*

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

Amir, Y., Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., and Ciarfella, P. (1995). The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4).

Babaoğlu, Ö., Davoli, R., and Montresor, A. (1995). Failure Detectors, Group Membership and View-Synchronous Communication in Partitionable Asynchronous Systems. TR UBLCS-95-18, Department of Conmputer Science, University of Bologna.

Birman, K. and Joseph, T. (1987). Exploiting virtual synchrony in distributed systems. In *11th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 123–138. ACM.

Birman, K., Schiper, A., and Stephenson, P. (1991). Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314.

Birman, K. and van Renesse, R. (1994). *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press.

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

Chockler, G., Huleihel, N., and Dolev, D. (1998). An adaptive totally ordered multicast protocol that tolerates partitions. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 237–246.

Chockler, G. V. (1997). An Adaptive Totally Ordered Multicast Protocol that Tolerates Partitions. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel.

De Prisco, R., Lampson, B., and Lynch (1997). Revisiting the Paxos algorithm. In Mavronicolas, M. and Tsigas, P., editors, *11th International Workshop on Distributed Algorithms (WDAG)*, pages 111–125, Saarbrucken, Germany. Springer Verlag. LNCS 1320.

Dolev, D., Friedman, R., Keidar, I., and Malki, D. (1996). Failure Detectors in Omission Failure Environments. TR 96-13, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel. Also Technical Report 96-1608, Department of Computer Science, Cornell University.

Dolev, D., Friedman, R., Keidar, I., and Malki, D. (1997). Failure detectors in omission failure environments. In *16th ACM Symposium on Principles of Distributed Computing (PODC)*, page 286. Brief announcement.

Dolev, D. and Malkhi, D. (1996). The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4).

Dolev, D. and Malki, D. (1995). The design of the Transis system. In Birman, K. P., Mattern, F., and Schipper, A., editors, *Theory and Practice in Distributed Systems: International Workshop*, pages 83–98. Springer Verlag. LNCS 938.

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

Fekete, A., Gupta, D., Luchangco, V., Lynch, N., and Shvartsman, A. (1996). Eventually-serializable data services. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 300–309.

Fekete, A., Lynch, N., and Shvartsman, A. (1997). Specifying and using a partionable group communication service. In *16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–62.

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

Friedman, R., Keidar, I., Malki, D., Birman, K., and Dolev, D. (1995). Deciding in Partitionable Networks. TR 95-16, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel. Also Cornell University TR95-1554.

Friedman, R. and van Renesse, R. (1995). Strong and Weak Virtual Synchrony in Horus. TR 95-1537, dept. of Computer Science, Cornell University.

Friedman, R. and van Renesse, R. (1997). Packing messages as a tool for boosting the performance of total ordering protocols. In *6th IEEE International Symposium on High Performance Distributed Computing*. Also available as Technical Report 95-1527, Department of Computer Science, Cornell University.

Hadzilacos, V. and Toueg, S. (1993). Fault-tolerant broadcasts and related problems. In Mullender, S., editor, *chapter in: Distributed Systems*. ACM Press.

Hayden, M. and van Renesse, R. (1996). Optimizing Layered Communication Protocols. Technical Report TR96-1613, Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA.

Kaashoek, M. F. and Tanenbaum, A. S. (1996). An evaluation of the Amoeba group communication system. In *16th International Conference on Distributed Computing Systems (ICDCS)*, pages 436–447.

Keidar, I. (1994). A Highly Available Paradigm for Consistent Object Replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel. Also Institute of Computer Science, The Hebrew University of Jerusalem Technical Report CS95-5.

Keidar, I. and Dolev, D. (1996). Efficient message ordering in dynamic networks. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–76.

Keidar, I. and Dolev, D. (1998). Increasing the resilience of distributed and replicated database systems. *Journal of Computer and System Sciences special issue with selected papers from ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS) 1995*, 57(3):309–324.

Lamport, L. (1989). The part-time parliament. TR 49, Systems Research Center, DEC, Palo Alto.

Lamport, L. (78). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Malki, D. (1994). *Multicast Communication for High Avalaibility*. PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem.

Malloth, C. P., Felber, P., Schiper, A., and Wilhelm, U. (1995). Phoenix: A toolkit for building fault-tolerant, distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*.

Mann, T., Hisgen, A., and Swart, G. (1989). An Algorithm for Data Replication. Technical Report 46, DEC Systems Research Center.

Moser, L. E., Amir, Y., Melliar-Smith, P. M., and Agarwal, D. A. (1994). Extended virtual synchrony. In *14th International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65. Full version: technical report ECE93-22, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA.

Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., Budhia, R. K., and Lingley-Papadopoulos, C. A. (1996). Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4).

Moser, L. E., Melliar-Smith, P. M., and Agrawala, V. (1993). Asynchronous fault-tolerant total ordering algorithms. *SIAM Journal on Computing*, 22(4):727–750.

Peleg, D. and Wool, A. (1995). Availability of quorum systems. *Inform. Comput.*, 123(2):210–223.

Powell, D. (1991). *Delta-4: A Generic Architecture for Dependable Distributed Computing.* Springer Verlag.

Pu, C. and Leff, A. (1991). Replica control in distributed systems: An asynchronous approach. In *ACM SIGMOD International Symposium on Management of Data.*

Schneider, F. B. (1990). Implementing fault tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319.

Skeen, D. (1982). A quorum-based commit protocol. In *6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 69–80.

van Renesse, R., Birman, K. P., and Maffeis, S. (1996). Horus: A flexible group communication system. *Communications of the ACM*, 39(4).

Whetten, B., Montgomery, T., and Kaplan, S. (1995). A high perfomance totally ordered multicast protocol. In Birman, K. P., Mattern, F., and Schipper, A., editors, *Theory and Practice in Distributed Systems: International Workshop*, pages 33–57. Springer Verlag. LNCS 938.

Yeger Lotem, E., Keidar, I., and Dolev, D. (1997). Dynamic voting for consistent primary components. In *16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 63–71.