Extended Virtual Synchrony

L. E. Moser, Y. Amir, P. M. Melliar-Smith, D. A. Agarwal
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106

Abstract. We formulate a model of extended virtual synchrony that defines a group communication transport service for multicast and broadcast communication in a distributed system. The model extends the virtual synchrony model of the Isis system to support continued operation in all components of a partitioned network. The significance of extended virtual synchrony is that, during network partitioning and remerging and during process failure and recovery, it maintains a consistent relationship between the delivery of messages and the delivery of configuration changes across all processes in the system and provides well-defined self-delivery and failure atomicity properties. We describe an algorithm that implements extended virtual synchrony and construct a filter that reduces extended virtual synchrony to virtual synchrony.

1 Introduction

In many applications in distributed systems messages must be disseminated to multiple destinations. To achieve better performance, protocols have been developed to exploit the multicast or broadcast capabilities of existing local-area network hardware [1, 3, 5, 9, 11, 13]. The process group paradigm [7] is a useful and appropriate addressing mechanism for multicast and broadcast communication.

Within the process group paradigm, virtual synchrony [4, 5, 6, 14] ensures that processes perceive process failures and other configuration changes as occurring at the same logical time. The model of virtual synchrony handles omission faults and fail-stop faults, and regards recovered processes as new processes. When network partitioning occurs, the virtual synchrony model also ensures that processes in at most one connected component of the network, the primary component, are able to make progress; processes in the other components of the network are blocked.

Unfortunately, if a process fails and can recover with stable storage intact, then inconsistenicies can arise. Consider, for example, the failure of a process that was responsible for deciding the order of messages and informing other processes of that order. It may decide an order and deliver messages locally in that order but fail to communicate that order to other processes. After removing the failed process from the configuration, the other processes may determine an order without knowing the order chosen by the failed process. If the failed process can recover with stable storage intact and if the contents of its stable storage can be affected by the order of delivery of messages, the model of virtual synchrony must be extended.

Gateways, bridges and wireless communication increase the probability of network partitioning, which may also result in inconsistencies. For example, if the process responsible for determining the order of messages becomes detached, it may continue to order and deliver messages locally after it has become detached but before it learns that it has become detached. The order in which it delivers messages before becoming detached may be inconsistent with the order in which other processes deliver messages; a problem can arise if a detached process can resume operation and remerge with the primary component. The extended virtual synchrony model guarantees that processes in all components of a partitioned network have a consistent, though perhaps incomplete, history of the system.

Moreover, in some applications it is not acceptable to block processes that are not in the primary component. The application should be allowed to determine which processing, if any, is appropriate while the network is partitioned. To illustrate this point, we present the following examples:

- An airline reservation system must continue to sell tickets even if the system becomes partitioned. Airlines have devised heuristics for use in non-primary components, based only on local data, that aim to maximize the number of tickets that can be sold while minimizing the risk of overbooking.

- An ATM machine, operating in a fully connected system, records each transaction in its database, checking that cumulative withdrawals do not exceed the account balance. When operating in a non-primary component, however, it consults a small database to authorize a withdrawal without checking for cumulative withdrawals at different locations, and delays posting the transaction until the system becomes reconnected.

This work was supported by the National Science Foundation, Grant No. NCR-9016361, by the Advanced Research Project Agency, Grant No. N00174-93-K-0097, by Rockwell CMC through the State of California MICRO program, Grant No. 92-101, and by the United States-Israel Binational Science Foundation, Grant No. 92-00189.
The address of Y. Amir is Computer Science Department, The Hebrew University of Jerusalem, 91904, Israel.

- A radar system combines a number of sensors, as well as a number of displays, in different locations. The most accurate available information, obtained from the sensor with the best view should be displayed to the operator. In the case of a network partition, however, it is better to display lower quality information from the connected sensors than to do nothing.

In the design of the Totem protocol [3, 12], based on our experience with the Trans and Total protocols [11] and the Transis system [1, 2], we have extended the virtual synchrony model [4, 5, 6] of the Isis system to handle network partitioning and remerging, as well as process failure and recovery. Extended virtual synchrony establishes a consistent relationship between delivery of messages and delivery of configuration changes across all processes in the system, and provides well-defined self-delivery and failure atomicity properties.

2 The Model and Services Provided

A *distributed system* is a finite set of processes that communicate over a network by sending messages. Each of the processes in the system has a unique identifier. A process may fail and may subsequently recover after an arbitrary amount of time with its stable storage intact. When a process recovers, it has the same identifier as before the failure. The network may partition into some finite number of *components*. The processes in a component can receive messages broadcast by other processes in the same component, but processes in two different components are unable to communicate with each other. Two or more components may subsequently merge to form a larger component.

Each process executes a low-level membership algorithm to determine the processes that are members of its component. This membership, together with a unique identifier, is called a *configuration*. The membership algorithm ensures that all processes in a configuration agree on the membership of that configuration. The application is informed of changes in the configuration by the delivery of configuration change messages.

Each process also executes a reliable broadcasting and ordering algorithm that associates an ordinal number with each message. These ordinals impose a total order on messages broadcast within a configuration. Processes deliver messages to the application in the order imposed by these ordinal numbers, an ordering that preserves causality. As an alternative to the total ordering algorithm, we can consider an ordering algorithm that only imposes a partial order on messages.

We distinguish between *receipt* of a message over the communication medium, which may be out of order, and *delivery* of a message to the application, which may be delayed until prior messages in the order have been delivered. Three message delivery services are defined:

- Causal delivery, defined in the context of network partitioning and remerging (cbcast in Isis)
- Agreed delivery, which guarantees a total order of message delivery within each component and allows a message to be delivered as soon as all of its predecessors in the total order have been delivered (abcast in Isis)
- Safe delivery, which guarantees that, if any process within a component delivers a message, then that message has been received and will be delivered by every other process in that component unless that process fails (all-stable abcast in Isis).

Causal delivery applies only to messages broadcast in the same configuration and does not extend back to prior configurations. Agreed and safe delivery impose severe requirements on the algorithms in the presence of network partitioning and remerging and of process failure and recovery. Process p guarantees to deliver every message broadcast for delivery in agreed order in configuration c that precedes the configuration change message delivered by p to terminate c. Delivery in safe order is even more demanding because it guarantees, in addition, that a message delivered in safe order by p will be delivered by every other process in c unless that process fails. In this paper we focus on safe messages.

To achieve safe delivery in the presence of network partitioning and remerging and of process failure and recovery, the extended virtual synchrony algorithm presents to the application two types of configurations. In a *regular configuration* new messages are broadcast and delivered. In a *transitional configuration* no new messages are broadcast but the remaining messages from the prior regular configuration are delivered. Those messages did not satisfy the safe or causal delivery requirements in the regular configuration and, thus, could not be delivered in that configuration.

A regular configuration may be immediately followed by several transitional configurations (one for each component of the partitioned network) and may be immediately preceded by several transitional configurations when several components merge together. A transitional configuration, in contrast, is immediately followed by a single regular configuration and is immediately preceded by a single regular configuration. A transitional configuration consists of the members of the next regular configuration that have the same preceding regular configuration. Messages can be delivered as safe in a transitional configuration even though they cannot be delivered as safe in the preceding regular configuration, so long as the application is informed of the configurations in which the messages are delivered. It is then up to the application to determine how to proceed with this information.

Each process in a transitional or regular configuration delivers a *configuration change message* to the application to terminate the prior configuration and initiate the new configuration. Delivery of a configuration change message that initiates a new configuration follows delivery of every message in the configuration that it terminates and precedes delivery of every message in the configuration that it initiates. The configuration change message that initiates a transitional configuration defines the membership within which it is possible to guarantee safe delivery of the remaining messages of the prior regular configuration.

For a process p that is a member of a regular configuration c, we define $\text{trans}_p(c)$ to be the transitional configuration that follows c at p, if such a configuration exists. For a process p that is a member of a transitional configuration c, $\text{trans}_p(c) = c$. For a process p that is a member of a transitional configuration c, we define $\text{reg}_p(c)$ to be the regular configuration that immediately precedes c. For a process p that is a member of a regular configuration c, $\text{reg}_p(c) = c$. We define $\text{com}_p(c)$ to be either one of the configurations $\text{reg}_p(c)$ or $\text{trans}_p(c)$. We use c to refer to a single specific configuration. If both p and q are members of c, then $\text{reg}_p(c) = \text{reg}_q(c)$. However, $\text{trans}_p(c)$ is not necessarily equal to $\text{trans}_q(c)$ and, thus, $\text{com}_p(c)$ is not necessarily equal to $\text{com}_q(c)$.

The specification of extended virtual synchrony is defined in terms of four types of events:

- $\text{deliver\_conf}_p(c)$: process p delivers a configuration change message initiating configuration c, where p is a member of c
- $\text{send}_p(m, c)$: process p sends (originates) message m while p is a member of configuration c
- $\text{deliver}_p(m, c)$: process p delivers message m while p is a member of configuration c
- $\text{fail}_p(c)$: process p fails while p is a member of configuration c.

The $\text{fail}_p(c)$ event is the actual failure of process p in configuration c and is distinct from a $\text{deliver\_conf}_q(c')$ event that removes p from configuration c. After a $\text{fail}_p(c)$ event, process p may remain failed forever or may recover with a $\text{deliver\_conf}_p(c'')$ event, where the membership of c'' is $\{p\}$.

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

2.1 The Extended Virtual Synchrony Model

The model of extended virtual synchrony consists of Specifications 1-7 below, which are expressed in terms of the partial order relation, $\rightarrow$, and the total order function, ord. The causal delivery requirements, given by Specification 5, apply only to messages sent (originated) within a single configuration.

Specifications 1-5 are illustrated in Figures 1-5. Specifications 6 and 7 are more difficult to depict and so are not shown. In these figures vertical lines correspond to processes, an open circle represents an event that is assumed to exist, a star represents an event that is asserted to exist, a light edge without an arrow represents a precedes relation that holds because of some other specification, a medium edge with an arrow represents a precedes relation that is assumed to hold, a heavy edge with an arrow represents a precedes relation that is asserted to hold, and a cross through an event (relation) indicates that the event (relation) does not occur.

In these specifications when we write "there exists $\text{send}_p(m, c)$" we mean that there exist a process p, a message m and a configuration c such that process p sends message m in configuration c and, similarly, for "there exists $\text{deliver}_p(m, c)$" and "there exists $\text{deliver\_conf}_p(m, c)$". Moreover, when we write "$\text{deliver}_p(m,\text{com}_p(c))$" we mean "$\text{deliver}_p(m,\text{reg}_p(c))$" or "$\text{deliver}_p(m,\text{trans}_p(c))$".

Basic Delivery

Specification 1.1 requires that the $\rightarrow$ relation is a partial order relation (reflexive,* anti-symmetric and transitive), and Specification 1.2 requires that the events within a single process are totally ordered by the $\rightarrow$ relation. Specification 1.3 requires that the sending of a message precedes its delivery, and that the delivery occurs in the configuration in which the message was sent or in an immediately following transitional configuration. Specification 1.4 asserts that a given process does not send, or deliver, the same message in two different configurations and that two different processes do not send the same message.

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

1.2. If there exists an event e that is $\text{send}_p(m, c)$, $\text{deliver}_p(m, c)$, $\text{fail}_p(c)$ or $\text{deliver\_conf}_p(c)$ and an event e' that is $\text{send}_p(m', c')$, $\text{deliver}_p(m', c')$, $\text{fail}_p(c')$ or $\text{deliver\_conf}_p(c')$, then $e \rightarrow e'$ or $e' \rightarrow e$.

1.3. If there exists $\text{deliver}_p(m, c)$, then there exists $\text{send}_q(m,\text{reg}_q(c))$ such that $\text{send}_q(m,\text{reg}_q(c)) \rightarrow \text{deliver}_p(m, c)$.

1.4. If there exists $\text{send}_p(m, c)$, then $c = \text{reg}_p(c)$ and there does not exist $\text{send}_p(m, c')$, where $c \neq c'$, or $\text{send}_q(m, c'')$, where $p \neq q$. Moreover, if there exists $\text{deliver}_p(m, c)$, then there does not exist $\text{deliver}_p(m, c')$, where $c \neq c'$.

Delivery of Configuration Changes

Specification 2.1 requires that, if a process is a member of a configuration and does not install or does not remain a member of that configuration, then the other processes install a new configuration. In particular, this means that if the process fails, then the other processes will detect the failure and install a new configuration. Specification 2.2 states that at any moment a process is a member of a unique configuration whose events are delimited by the configuration change event(s) for that configuration. Specifications 2.3 and 2.4 assert that an event that precedes (follows) delivery of a configuration change by one process must also precede (follow) delivery of that configuration change by other processes.

2.1. If there exists $\text{deliver\_conf}_p(c)$, there does not

*The $\rightarrow$ relation could have been defined to be irreflexive but, to conform to the standard mathematical definition of a partial order, we define the $\rightarrow$ relation to be reflexive.

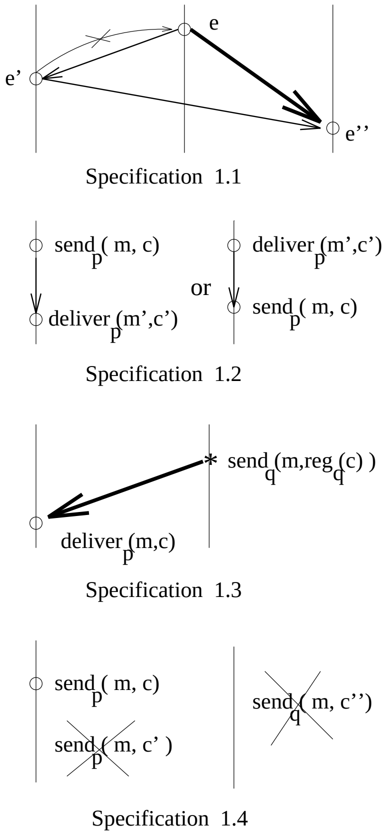


Figure 1: *Basic Delivery Specifications.*

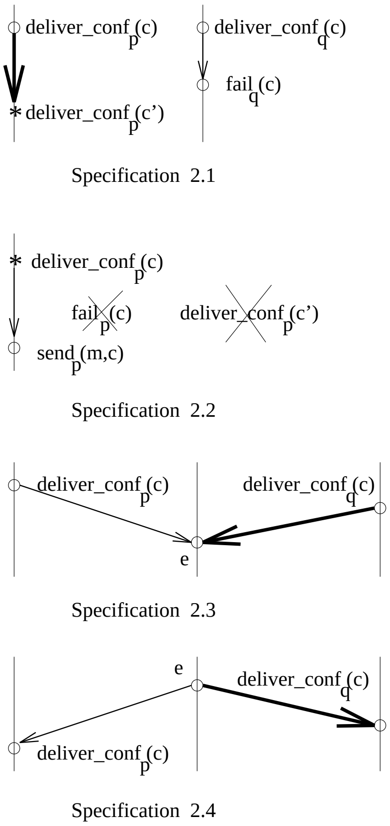


Figure 2: *Configuration Change Specifications.*

exist $\text{fail}_p(c)$, there does not exist $\text{deliver\_conf}_p(c')$ such that $\text{deliver\_conf}_p(c) \rightarrow \text{deliver\_conf}_p(c')$, where $c \neq c'$, and if q is a member of c, then there exists $\text{deliver\_conf}_q(c)$, there does not exist $\text{fail}_q(c)$ and there does not exist $\text{deliver\_conf}_q(c'')$ such that $\text{deliver\_conf}_q(c) \rightarrow \text{deliver\_conf}_q(c'')$, where $c \neq c''$.

2.2. If there exists an event e that is either $\text{send}_p(m, c)$ or $\text{deliver}_p(m, c)$ or $\text{fail}_p(c)$, then there exists $\text{deliver\_conf}_p(c)$ such that $\text{deliver\_conf}_p(c) \rightarrow e$ and there does not exist an event e' such that e' is $\text{fail}_p(c)$ or $\text{deliver\_conf}_p(c')$ and $\text{deliver\_conf}_p(c) \rightarrow e' \rightarrow e$, where $e \neq e'$ and $c \neq c'$.

2.3. If there exist $\text{deliver\_conf}_p(c)$, $\text{deliver\_conf}_q(c)$ and e such that $\text{deliver\_conf}_p(c) \rightarrow e$, where $e \neq \text{deliver\_conf}_p(c)$, then $\text{deliver\_conf}_q(c) \rightarrow e$.

2.4. If there exist $\text{deliver\_conf}_p(c)$, $\text{deliver\_conf}_q(c)$ and e such that $e \rightarrow \text{deliver\_conf}_p(c)$, where $e \neq \text{deliver\_conf}_p(c)$, then $e \rightarrow \text{deliver\_conf}_q(c)$.

Self-Delivery

Specification 3 requires that each process delivers each message it sends, provided that it does not fail. This delivery may occur in a transitional configuration that consists of only the process that sent the message.

3. If there exist $\text{send}_p(m, c)$ and $\text{deliver\_conf}_p(c')$ such that $\text{send}_p(m, c) \rightarrow \text{deliver\_conf}_p(c')$, where $c' \neq \text{trans}_p(c)$, and there does not exist $\text{fail}_p(\text{com}_p(c))$, then there exists $\text{deliver}_p(m, \text{com}_p(c))$.

Failure Atomicity

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

4. If there exist $\text{deliver\_conf}_p(c)$, $\text{deliver\_conf}_p(c''')$, $\text{deliver\_conf}_q(c)$, $\text{deliver\_conf}_q(c''')$ and $\text{deliver}_p(m, c)$ such that $\text{deliver\_conf}_p(c) \rightarrow \text{deliver\_conf}_p(c''')$, where $c \neq c'''$, and there does not exist $\text{deliver\_conf}_p(c')$ such that $\text{deliver\_conf}_p(c) \rightarrow \text{deliver\_conf}_p(c') \rightarrow \text{deliver\_conf}_p(c''')$, where $c \neq c'$ and $c' \neq c'''$, and there does not exist $\text{deliver\_conf}_q(c'')$ such that $\text{deliver\_conf}_q(c) \rightarrow \text{deliver\_conf}_q(c'') \rightarrow$

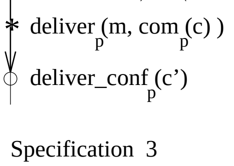
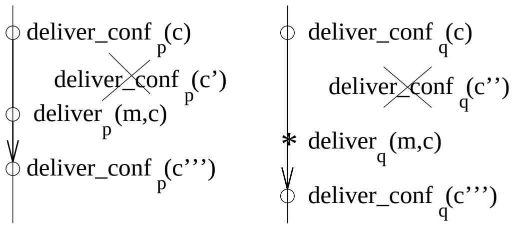
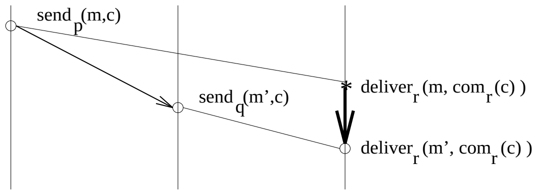


Figure 3: *Self Delivery Specification*



Specification 4

Figure 4: *Failure Atomicity Specification*



Specification 5

Figure 5: *Causal Delivery Specifications.*

deliver_conf$_q(c''')$, where $c \neq c''$ and $c'' \neq c'''$, then there exists deliver$_q(m,c)$.

Causal Delivery

Unlike other researchers, we model causality so that it is local to a single configuration and is terminated by a membership change. Simpler formulations of causality are not appropriate when a network may partition and remerge or when a process may fail and restart with stable storage intact and with the same identifier.

The causal relationship between messages is expressed in Specification 5 as a precedes relation between the sending of two messages in the same configuration. This precedes relation is contained in the transitive closure of the precedes relations established by Specifications 1.1-1.3.

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

5. If there exist send$_p(m,c)$, send$_q(m',c)$ and deliver$_r(m'$,com$_r(c))$ such that send$_p(m,c) \rightarrow$ send$_q(m',c)$, then there exists deliver$_r(m$,com$_r(c))$ such that deliver$_r(m$,com$_r(c)) \rightarrow$ deliver$_r(m'$,com$_r(c))$.

Totally Ordered Delivery

The following specifications constrain the definition of the ord function. Specification 6.1 requires the total order to be consistent with the partial order. Specification 6.2 asserts that processes deliver configuration change messages for the same configuration at the same logical time and that they deliver the same message at the same logical time. Specification 6.3 requires that processes deliver messages in order except that, in the transitional configuration, there is no obligation to deliver messages sent by processes not in the transitional configuration.

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

6.2. If there exist events e and e' that are either deliver_conf$_p(c)$ and deliver_conf$_q(c)$ or deliver$_p(m,c)$ and deliver$_q(m,c')$, then ord$(e) =$ ord(e').

6.3. If there exist deliver$_p(m$,com$_p(c))$, deliver$_p(m'$, com$_p(c))$, deliver$_q(m',c')$, send$_r(m$,reg$_r(c'))$ such that ord(deliver$_p(m$,com$_p(c))) <$ ord(deliver$_p(m'$,com$_p(c)))$ and r is a member of c', then there exists deliver$_q(m$,com$_q(c'))$.

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

Safe Delivery

Specification 7.1 requires that, if any process delivers a message in a configuration, then each process in that configuration delivers the message unless that process fails. Specification 7.2 asserts that, if any process delivers a safe message in a regular configuration, then all processes in that configuration deliver configuration change messages for that configuration.

7.1. If there exists deliver$_p(m,c)$ for a safe message m, then for all members q of c there exists deliver$_q(m$,com$_q(c))$ or fail$_q$(com$_q(c))$.

7.2. If there exists deliver$_p(m$,reg$_p(c))$ for a safe message m, then for all members q of reg$_p(c)$ there exists deliver_conf$_q$(reg$_p(c))$.

Finally, note that the Basic Delivery Specification 1.2, when restricted to a single configuration, expresses causality of events within a single process. Also note that, if we modify Specification 5 by replacing send$_p(m,c)$ by deliver$_q(m,c)$, then the modified specification follows from the existing Specification 5 and Specification 1.3.

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

2.2 The Primary Component Model

The properties required of the history H of primary components are defined below, where C, C' and C'' represent primary components.

Uniqueness

The history H of primary components is totally ordered by the $\rightarrow$ relation.

1. If there exist deliver_conf$_p(C)$, deliver_conf$_q(C')$ in H, then deliver_conf$_p(C) \rightarrow$ deliver_conf$_q(C')$ or deliver_conf$_q(C') \rightarrow$ deliver_conf$_p(C)$.

Continuity

For each pair of consecutive primary components in the history H, at least one process is a member of both.

2. If there exist deliver_conf$_p(C)$, deliver_conf$_r(C'')$ in H and there does not exist deliver_conf$_q(C')$ in H such that deliver_conf$_p(C) \rightarrow$ deliver_conf$_q(C') \rightarrow$

deliver_conf$_r(C'')$, where $C \neq C'$ and $C' \neq C''$, then there exists a process s that is a member of both C and C''.

3 An Algorithm for Implementing Extended Virtual Synchrony

We now present an algorithm that implements extended virtual synchrony for safe delivery of totally ordered messages on top of the message transmission, membership, and total ordering algorithms. The Totem protocol [3] incorporates these algorithms and provides extended virtual synchrony. The steps of the extended virtual synchrony algorithm, executed by an individual process, are as follows.

1. In a regular configuration, this process sends and receives messages, holding in a message buffer any messages that it has received but cannot yet deliver. The process delivers a message as safe when it has delivered all of the messages that precede the message in the total order and has received acknowledgments for the message from all of the other processes in the configuration. An acknowledgment indicates that a process has received and will deliver the message unless it fails.

In a regular configuration, this process records that there are no processes to which it is obligated. A process p is *obligated* to a process q when p has transmitted an acknowledgment for a message m sent (originated) by q that enables another process to deliver m as safe. The set of processes to which p is obligated is referred to as its *obligation set*.

When this process has been informed by the underlying membership algorithm of the membership and identifier of a proposed new configuration, it commences to perform the following steps, which constitute the recovery algorithm.

2. Buffer or reject all new messages from the application until this process delivers a configuration change message for a regular configuration to the application. Buffer any messages received for the proposed new configuration.

3. Exchange information with each process of the proposed new configuration. In particular, each process supplies the identifier of its last regular configuration, the identifier of the last safe message it delivered, and its obligation set.

4.a. Determine the members of the proposed transitional configuration of this process, *i.e.* the members of the new regular configuration whose previous regular configuration is the same as the previous regular configuration of this process.

b. Determine the messages to be rebroadcast because some process in the proposed transitional configuration of this process has not acknowledged receipt of those messages.

5.a. Rebroadcast messages as required by Step 4.b and acknowledge receipt of such messages.

b. Continue Step 5.a until all processes in the proposed transitional configuration of this process acknowledge having received all of the rebroadcast messages.

c. If during Step 5.a this process acknowledges having received all of the rebroadcast messages, it includes the members of the proposed transitional configuration and their obligation sets in its obligation set.

6.a. Discard all messages, except those sent by a member of the obligation set of this process, that follow the first unavailable message in the total order. Such messages must be discarded because they may be causally dependent on an unavailable message. The obligation set includes all members of the proposed transitional configuration of this process.

b. Deliver to the application in order all of the rebroadcast messages that are safe in the preceding regular configuration up to but not including the first totally ordered message for which a predecessor in the total order is unavailable, or the first message for which safe delivery was requested but for which some process in the preceding regular configuration has not acknowledged receipt.

c. Deliver a first configuration change message that introduces the transitional configuration.

d. Deliver in order, from the remaining undelivered messages, all messages whose predecessors in the total order have been delivered, and all messages sent by a process in the obligation set of this process.

e. Deliver a second configuration change message to terminate the transitional configuration and install the new regular configuration reported by the underlying membership algorithm.

The parts of Step 6 are performed locally as an atomic action without communication with any other process. If a failure occurs during execution of the recovery algorithm, then the membership algorithm is invoked and the recovery algorithm is restarted at Step 2.

3.1 An Example of Configuration Changes and Message Delivery

Consider the example shown in Figure 6. Here a regular configuration containing processes p, q and r partitions and p becomes isolated while q and r merge into a regular configuration with processes s and t. Processes q and r deliver two configuration change messages, one to shift from the old regular configuration $\{p,q,r\}$ to the transitional configuration $\{q,r\}$ and the other to shift from the transitional configuration $\{q,r\}$ to the new regular configuration $\{q,r,s,t\}$.

Processes q and r may not be able to deliver all of the messages broadcast in the regular configuration $\{p,q,r\}$. In particular, they cannot deliver any message for which the causal or safe delivery requirement for $\{p,q,r\}$ is not satisfied.

If process p sends message m after sending message l but q and r did not receive l before a configuration change occurred, then q cannot deliver m because its causal predecessor l is not available.

By the self-delivery property (Specification 3), q and r must each deliver the messages they themselves sent in $\{p,q,r\}$. Of course, each process q and r has its own messages and also any messages that causally precede its own messages, since it must have delivered such messages before it sent its own messages.

After the message exchange for the transitional con-

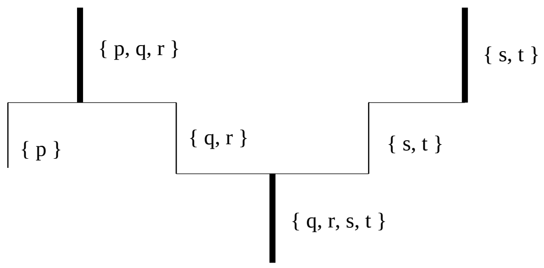


Figure 6: *Configuration Changes and Message Delivery.*

figuration $\{q, r\}$ has been completed, both q and r have all messages sent by q or r and all the causal predecessors of such messages. Furthermore, all such messages are safe in $\{q, r\}$ and, consequently, can be delivered in the transitional configuration.

If process r sends message n for safe delivery but does not receive an acknowledgment for n from both p and q before a configuration change occurs, then r cannot deliver n in the regular configuration $\{p, q, r\}$. If, however, r receives an acknowledgment for n from q, then r can deliver n in the transitional configuration $\{q, r\}$.

3.2 Proof that the Algorithm Satisfies Extended Virtual Synchrony

Specification 1.1 states that the $\rightarrow$ relation is a partial order. The reflexive property is a matter of definition. The transitive and acyclic properties are assumptions that we are making about the real world. Specification 1.2 expresses the fact that a process has a single thread of control. Specifications 1.3 and 1.4 follow from the underlying broadcast mechanisms.

Specifications 2.1-2.4 follow from the underlying membership algorithm.

Specification 3 requires that a process delivers its own messages, provided that it does not fail. In particular, when a process considers the undelivered messages in Step 6 of the extended virtual synchrony recovery algorithm, no message sent by any member of the transitional configuration is discarded on the grounds that it is causally dependent on an unavailable message. All of the preceding messages must have been available to the process that sent the message and, thus, are available to all members of the transitional configuration after the message exchange.

Specification 4 requires that processes deliver the same set of messages in a regular configuration and the same set of messages in a transitional configuration. After the message exchange in Step 5 of the extended virtual synchrony recovery algorithm, all processes in the transitional configuration have the same set of messages and apply the same algorithm to determine message delivery in the regular and transitional configurations.

Specification 5 follows immediately if m' is delivered in a regular configuration. If m' is delivered in a transitional configuration, then q is a member of that configuration or of the obligation set. Since $\mathrm{send}_p(m, c) \rightarrow \mathrm{send}_q(m', c)$, either $p = q$ or m was delivered by q before q sent m' and, thus, m is safe in c. In either case, m is delivered before m' in the regular or transitional configuration.

Specifications 6.1 and 6.2 follow from the definition of the ord function and from the consistency provided by Step 6 of the extended virtual synchrony recovery algorithm and by the message total ordering algorithm. In addition, Specification 6.1 depends on the fact that a process has a single thread of control.

Specification 6.3 follows by an argument similar to that for Specification 3. In Step 6.a of the extended virtual synchrony recovery algorithm, messages from processes not in the transitional configuration may be dropped, but messages from members of the transitional configuration are delivered in order.

Specification 7.1 is obvious if all processes complete the extended virtual synchrony recovery algorithm. If, however, further processes fail or a further partition occurs during the recovery algorithm, more care is required. Some processes may not complete the recovery algorithm but may instead receive a further membership change from the underlying membership algorithm, causing them to restart the recovery algorithm. If such a process has acknowledged receipt of all of the rebroadcast messages, it is possible that some other process may have completed the recovery algorithm and installed the next regular configuration before the failure occurred. The other process may have delivered messages as safe in the transitional configuration, relying on the acknowledgment supplied by this process. The concept of obligation ensures that these messages are indeed delivered by all of the processes needed to satisfy the safe delivery requirement.

Specification 7.2 follows directly from Step 6.e of the extended virtual synchrony recovery algorithm.

Termination Property

Note that the termination of the recovery algorithm is dependent on the termination of the membership algorithm. The underlying membership algorithm will eventually terminate if it has the property that, if the next proposed regular configuration is not installed within a bounded time, then the membership of that configuration is reduced. The Totem protocol and the Transis system preserve extended virtual synchrony and contain a membership algorithm that terminates within a bounded time.

4 The Virtual Synchrony Model

We now summarize Birman's model of virtual synchrony, as it is presented in [6] where more discussion and details can be found. We then show in Section 5 how virtual synchrony can be implemented on top of extended virtual synchrony. This model of virtual synchrony is based on Lamport's causality relation, $\rightarrow$, defined in [10], *i.e.* the transitive closure of

- $e \rightarrow e'$, where e and e' are local to a process
- $\mathrm{send}(m) \rightarrow \mathrm{deliver}(m)$

The events local to a process are $\mathrm{send}(m)$, $\mathrm{deliver}(m)$ and stop. In addition, the virtual synchrony model has the group events: $\mathrm{view}_i(g)$, $\mathrm{cbcast}(g, m)$ and $\mathrm{abcast}(g, m)$, where g is a group, i is a process and m is a message.

A history H is said to be *complete* if

C1. For each event $e' \in H$ and for all $e \to e'$, $e \in H$.

C2. For each send(m) $\in H$, there is a corresponding deliver(m) $\in H$.

C3. Each multicast message m, that is delivered by a process in view g^x, is delivered by all other members of g^x, where x denotes the xth instance of group g.

A complete history H is said to be *legal* if it satisfies the following constraints:

L1. Each event $e \in H$ can be labelled with a global time, time(e), that respects the causal order of events, *i.e.* for any two events e and e', $e \to e'$ implies time(e) $<$ time(e').

L2. Distinct events of the same process have distinct times.

L3. Membership change events for the same view but distinct processes have the same logical time, *i.e.* $\text{time}(\text{view}_i(g^x)) = \text{time}(\text{view}_j(g^x))$.

L4. Deliver events of a multicast message m occur in the same view g^x for each process that delivers m, *i.e.* for each process i that delivers m the most recent membership change event preceding $\text{deliver}_i(m)$ is $\text{view}_i(g^x)$.

L5. For any two events $\text{deliver}_i(m)$ and $\text{deliver}_j(m)$ of an abcast message m, $\text{time}(\text{deliver}_i(m)) = \text{time}(\text{deliver}_j(m))$.

Extend(H) is defined to be the set of histories obtained by extending the local process histories within the history H by appending any missing deliver and view events that correspond to unpaired send, cbcast, abcast and view events in H.

Failure of a process is modeled by the distinguished final event, *stop*. The history of a failed process is extended by prepending the missing events before the stop event, but after any other events executed by the failed process prior to the failure.

A system execution is *acceptable* if, for any history H, there exists a history $H' \in$ extend(H) that is correct and legal.

A system is *virtually synchronous* if deliver(m) and view(g) events appear to occur simultaneously in the processes in which they occur.

4.1 The Failure Model

Birman assumes that failures respect the fail-stop model, and adopts a primary partition model in which at most one primary partition† is permitted to continue execution. A membership service notifies members of the primary partition when failures occur. The failed process is then removed from the primary partition. If a failed process subsequently recovers and reconnects to the primary partition, it does so with a new identifier.

A failure appears as a stop event that satisfies the following properties:

1. The membership service behaves like a single, continuously operational process. If a partition occurs, progress is permitted in only one partition, if any.
2. A failed process will be dropped from any groups to which it belongs, *i.e.* if $P_i[t]$ = stop, then there exists $t' > t$ such that, for all groups g, $P_i \in g[t] \Rightarrow P_i \notin g[t']$.
3. After a process has been observed to fail, no additional messages will be received from it.

4.2 Multicast Delivery Guarantees

A *uniform multicast* is a multicast m such that if any process delivers m in g^x then, even if that process fails, all processes deliver m in g^x. A multicast m that does not guarantee this uniformity property is a *non-uniform multicast.*

5 An Algorithm for Implementing Virtual Synchrony on Top of Extended Virtual Synchrony

We now provide an algorithm for implementing virtual synchrony on top of our basic model, the extended virtual synchrony algorithm, and a primary component algorithm (Figure 7). We construct a filter on a system that maintains extended virtual synchrony and show that all of the runs produced by this filter are acceptable executions according to the virtual synchrony model.

The primary component algorithm receives configuration change messages from the membership algorithm. It delivers these messages to the application with an indication as to whether the new configuration is a primary component. A simple primary component algorithm is easily constructed; we are currently developing an algorithm that has a greater probability of finding a primary component and thereby reduces the risk that all processes will be blocked.

The filter runs locally at a process within a configuration and is defined as follows:

1. Upon receiving a configuration change message for a transitional configuration $\text{trans}_p(c)$, mask this event and transform all $\text{deliver}_p(m,\text{trans}_p(c))$ events into $\text{deliver}_p(m,\text{reg}_p(c))$ events until the next deliver_config event for a regular configuration is received.
2. Upon receiving a configuration change message for a regular configuration that is not a primary component, block, *i.e.* don't accept any messages from the application for sending and discard any messages or configuration changes received until this process becomes a member of the primary component.
3. For a process in the primary component, upon receiving a configuration change message for a regular configuration that is a primary component and that merges a non-primary component containing several processes into the primary component, split the delivery of the single configuration change message into multiple events each of which merges one process at a time into the primary component in a deterministic order (such as lexicographical order).

†We use the term "component" to refer to a set of processes that can communicate among themselves and that are not able to communicate with processes in other components, and "partition" to refer to the collection of components that comprise the system. Thus, a primary partition in Birman's terminology is a primary component in our terminology.

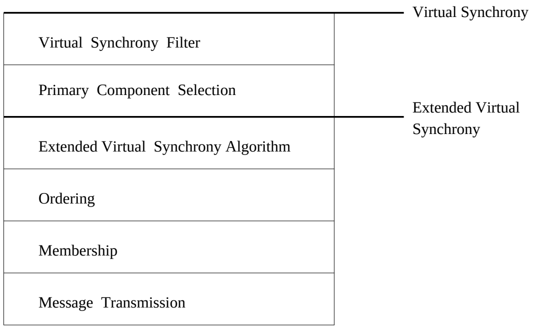


Figure 7: *Virtual Synchrony and Extended Virtual Synchrony.*

4. For a process in a non-primary component, upon receiving a configuration change message for a regular configuration that is a primary component, merge the processes in the non-primary component into the primary component, generating configuration change events as required in Rule 3.

In the extended virtual synchrony model a process that fails and recovers installs a singleton configuration. This singleton configuration is not the primary component and, thus, is blocked by the filter because of Rule 2 until the process is merged with the primary component in Rule 4.

In the extended virtual synchrony model there is no change in identifier of a resumed process; however, in the virtual synchrony model a resumed process has a new identifier. We can easily accommodate this in Rule 4 of the filter by giving a new identifier to a process on being merged into the primary component.

5.1 Proof that the Algorithm Satisfies Virtual Synchrony

A run produced by this filter can be completed using the extend mechanism of the virtual synchrony model. We now show that the completed run is legal. Our ord function corresponds to Birman's time function; both provide virtual or logical event ordering.

Property C1 corresponds to Specifications 1.3, 1.4, 2.2 and 5.

Property C2 is achieved by Specification 3 and the extend mechanism which yields a complete history. If there were a $\text{fail}_p(c)$ event in the filtered history, then the extend mechanism would add all of the $\text{deliver}_p(m, c)$ events that correspond to unmatched $\text{send}_p(m, c)$ events prior to this $\text{fail}_p(c)$ event.

Property C3 is achieved by Specification 4 and the extend mechanism if appropriately revised to exclude from the history messages sent by failed processes that were not delivered by one or more processes that do not fail.

Property L1 follows directly from our assumption of the ord function and Specification 6.1, if we assume that the events in L1 are distinct.

Property L2 follows from Specifications 1.1, 1.2 and 6.1.

Property L3 follows from Specification 6.2, where the $\text{view}_i(g^x)$ event corresponds to our $\text{deliver\_conf}_p(c)$ event.

Property L4 is achieved by first applying the extend mechanism to achieve a complete history. By Specifications 1.3 and 1.4, for each $\text{deliver}_p(m, c)$, there exists $\text{send}_q(m,\text{reg}_q(c))$, where $c = \text{reg}_p(c)$ or $\text{trans}_p(c)$ and $\text{reg}_p(c) = \text{reg}_q(c)$. By Specification 2.2, there exists $\text{deliver\_conf}_p(c)$ such that $\text{deliver\_conf}_p(c) \rightarrow \text{deliver}_p(m, c)$ and there does not exist $\text{deliver\_conf}_p(c')$, where $c \neq c'$, such that $\text{deliver\_conf}_p(c) \rightarrow \text{deliver\_conf}_p(c') \rightarrow \text{deliver}_p(m, c)$. Rule 1 of the filter masks all $\text{deliver\_conf}_p(\text{trans}_p(c''))$ events and transforms all $\text{deliver}_p(m,\text{trans}_p(c''))$ events into $\text{deliver}_p(m,\text{reg}_p(c''))$ events. Therefore, after the filter has been applied, message m is delivered in the view in which it was sent.

Property L5 follows from Specification 6.2.

5.2 Comparison of the Failure Models

The failure model of extended virtual synchrony, which allows network partitioning and remerging and also process failure and recovery with stable storage, is more general than the fail-stop model of virtual synchrony described in Section 4.1. It is possible to simulate fail-stop behavior in the extended virtual synchrony model by requiring a failed process to assume a new identity when it recovers.

The definition of a primary partition (component) is stated as Property 1 of the failure model of virtual synchrony. In that model as well as in our model an algorithm for maintaining a history of primary components may block.

Property 2 of the failure model of virtual synchrony is stronger than (does not follow from) Specification 2.1 of the extended virtual synchrony model. We allow a process to fail and recover sufficiently rapidly that it can be included in the next configuration, whereas the failure model of virtual synchrony requires the process to be excluded from that and all future configurations.

Property 3 of the failure model of virtual synchrony derives from Specification 2.2. After filtering and the delivery of a configuration change, no message is delivered that was sent by a process that was a member of the old configuration but not the new configuration, in particular because that process failed.

5.3 Comparison of the Multicast Properties

It is interesting to compare the different approaches used by virtual synchrony and extended virtual synchrony to achieve an approximation to the property that a message is not delivered unless it is delivered by all members of the configuration. Perfection is not possible as it would require common knowledge [8].

The virtual synchrony approach achieves this approximation in uniform multicast by extending the history using the extend mechanism, which assumes that the last few events in a failed process are lost forever and, thus, can impute delivery of a uniform multicast message to a failed process. This approach does not, of course, address systems that may partition and remerge or processes that may fail and restart with stable storage intact.

The extended virtual synchrony approach achieves this approximation in safe delivery, as defined by Specifications 7.1 and 7.2. It accepts that, for some messages, it may be impossible to determine whether a failed process has delivered them. The key mechanism of extended virtual synchrony is reduction in the size of the configuration. If it is impossible to determine whether a process will deliver a message, because of process failure or network partitioning, then a smaller transitional configuration is created, excluding that process. All processes in this smaller transitional configuration will deliver the message. Whether the more precise information provided by extended virtual synchrony is useful to an application program depends on the needs and sophistication of the application.

Another difference between the models is in the delivery of messages. Virtual synchrony requires in Property C1 that, for each message sent, some process delivers that message (not necessarily the one that sent it). In contrast, extended virtual synchrony requires in Specification 3 that each message is delivered by the process that sent it unless that process fails. The assumption of the virtual synchrony model is satisfied conceptually by extending the history using the extend mechanism, whereas the safe property of the extended virtual synchrony model ensures that the self-delivery requirement is satisfied by an actual history.

6 Conclusion

Extended virtual synchrony is a valuable abstraction for a distributed system. It maintains a consistent relationship between the delivery of messages and the delivery of configuration changes across all processes in a distributed system, even in the presence of network partitioning and remerging and of process failure and recovery with stable storage intact.

We have described an algorithm that implements extended virtual synchrony. This algorithm is currently operating in the Totem protocol at the University of California, Santa Barbara, and in the Transis system at the Hebrew University of Jerusalem.

We have also described a filter, running on top of extended virtual synchrony, that implements the Isis virtual synchrony model. This demonstrates that extended virtual synchrony does indeed extend virtual synchrony.

Acknowledgment. We wish to thank Danny Dolev for his insights and encouragement of this work.

References

[1] Y. Amir, D. Dolev, S. Kramer and D. Malki, "Transis: A communication sub-system for high availability," *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, Boston, MA (July 1992), pp. 76–84.

[2] Y. Amir, D. Dolev, S. Kramer and D. Malki, "Membership algorithms in broadcast domains," *Proceedings of the 6th International Workshop on Distributed Algorithms*, Haifa, Israel (November 1992), Lecture Notes in Computer Science 647, pp. 292-312.

[3] 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," *Proceedings of the IEEE 13th International Conference on Distributed Computing Systems*, Pittsburgh, PA (May 1993), pp. 551–560.

[4] K. P. Birman and T. A. Joseph, "Exploiting virtual synchrony in distributed systems," *Proceedings of the ACM Symposium on Operating System Principles* (1987), pp. 123-138.

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

[6] K. P. Birman, "Virtual synchrony model," In: Reliable Distributed Computing with the Isis Toolkit, IEEE Press.

[7] D. R. Cheriton and W. Zwaenepoel, "Distributed process groups in the V kernel," *ACM Transactions on Computer Systems* 3, 2 (May 1985), pp. 77–107.

[8] J. Y. Halpern and Y. Moses, "Knowledge and common knowledge in a distributed environment," *Journal of the ACM* 37, 3 (July 1990), pp. 549–587.

[9] M. F. Kaashoek and A. S. Tanenbaum, "Group communication in the Amoeba distributed operating system," *Proceedings of the IEEE 11th International Conference on Distributed Computing Systems* (May 1991), pp. 882–891.

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM* (July 1978), pp. 558–565.

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

[12] P. M. Melliar-Smith, L. E. Moser and D. A. Agarwal, "Ring-based ordering protocols," *Proceedings of the International Conference on Information Engineering*, Singapore (December 1991), pp. 882–891.

[13] L. L. Peterson, N. C. Buchholz and R. D. Schlichting, "Preserving and using context information in interprocess communication," *ACM Transactions on Computing Systems* 7, 3 (January 1989), pp. 217–246.

[14] A. Schiper and A. Sandoz, "Uniform reliable multicast in a virtually synchronous environment," *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, PA (May 1993), pp. 561–568.