

# The SecureRing Protocols for Securing Group Communication\*

Kim Potter Kihlstrom, L. E. Moser, P. M. Melliar-Smith

*Department of Electrical and Computer Engineering*

*University of California, Santa Barbara, CA 93106*

*kimk@alpha.ece.ucsb.edu, moser@ece.ucsb.edu, pmms@ece.ucsb.edu*

## Abstract

*The SecureRing group communication protocols provide reliable ordered message delivery and group membership services despite Byzantine faults such as might be caused by modifications to the programs of a group member following illicit access to, or capture of, a group member. The protocols multicast messages to groups of processors within an asynchronous distributed system and deliver messages in a consistent total order to all members of the group. They ensure that correct members agree on changes to the membership, that correct processors are eventually included in the membership, and that processors that exhibit detectable Byzantine faults are eventually excluded from the membership. To provide these message delivery and group membership services, the protocols make use of an unreliable Byzantine fault detector.*

## 1. Introduction

Group communication protocols provide a foundation on which fault-tolerant distributed systems can be built. By coordinating the delivery of messages, group communication protocols make it easier to keep replicated information consistent as it is updated in the presence of faults. Existing group communication protocols are resilient to crash faults,

---

\* Copyright 1998 IEEE. Published in the Proceedings of the Hawai'i International Conference on System Sciences, January 6-9, 1998, Kona, Hawaii.

Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions/IEEE Service Center/445 Hoes Lane/P.O. Box 1331/Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

Effort sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under grant number F30602-95-1-0048. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency and Rome Laboratory or the U.S. Government.

but few handle Byzantine faults. Byzantine faults can be caused by modifications to the programs of a group member following illicit access to, or capture of, a group member. They can also have been surreptitiously inserted earlier into the regular programs of a group member (a Trojan horse).

We describe here SecureRing, a suite of group communication protocols that provide protection against Byzantine faults. These protocols multicast messages to groups of processors within an asynchronous distributed system, impose a consistent total order on messages, and maintain consistent group memberships.

The approach adopted by SecureRing to protect against Byzantine faults is to optimize the performance for normal (fault-free) operation and to pay a performance penalty when a Byzantine fault is detected, which is expected to be rare. Protocols that require all messages to be digitally signed or that use extra rounds of message exchange incur a severe performance penalty during normal operation. In SecureRing, a group member embeds digests for the messages it has multicast into a signed token that it multicasts to reduce the performance penalty.

Developing protocols that are resistant to Byzantine faults is a difficult task, and starting with well-understood protocols makes that task somewhat easier. The SecureRing protocols presented here are inspired by the Totem single-ring protocols [1, 2, 11], which provide reliable, totally ordered delivery of messages despite processor crash faults and network partitioning faults. A logical token-passing ring is imposed on processors in a broadcast domain. The total order on messages is derived from a sequence number in the token. A flow control mechanism reduces message loss due to overflow of processor input buffers, and a membership protocol maintains consistent group memberships.

The problems of maintaining a consistent total order on messages and consistent group memberships are related to the problem of reaching consensus [4] and are impossible to solve in an asynchronous distributed system that is subject to faults. However, these problems can be solved if an unreliable fault detector<sup>†</sup> [3] is used. In [5] and [8] unreliable

---

<sup>†</sup> Popular terminology refers to these as unreliable failure detectors but, according to the ISO standard definitions of fault and failure, they should properly be called unreliable fault detectors.

fault detectors have been defined for environments that are subject to Byzantine faults, and consensus has been shown to be solvable in such environments using a fault detector, provided that at least  $\lceil(2n + 1)/3\rceil$  processors are correct in a system of  $n$  processors.

Reiter's Rampart system [14, 15] provides reliable and atomic group multicast in asynchronous distributed systems subject to process corruption. This is achieved by an echo strategy that requires, for each multicast to  $n$  destinations, the transmission of one unsigned multicast message,  $O(n)$  signed point-to-point messages, and one additional unsigned multicast message. The Rampart membership protocol makes use of a three-phase commit strategy. The network underlying Rampart provides an authenticated FIFO channel between each pair of processors.

The Trans/Total system [9] provides reliable totally ordered delivery of messages in asynchronous distributed systems. Byzantine fault-tolerant versions of the Total protocol are given in [10]. Four membership protocols [12] have been developed that operate on top of the Total protocol. Further work toward rendering the Trans/Total system resistant to Byzantine faults is being pursued in parallel with the work described here.

In [7] Malkhi and Reiter have described a multicast protocol that tolerates the malicious corruption of group members. Their method of chaining acknowledgments is based in part on the Trans protocol. Their protocol provides reliable multicasts, rather than the reliable totally ordered multicasts provided by the SecureRing protocols. It assumes a static set of processors, in contrast to SecureRing which supports dynamic membership changes.

## 2. System Model

We consider an asynchronous distributed system consisting of processors  $p_1, p_2, \dots$  that communicate via messages over a network that is completely connected. We use  $p, q, r, \dots$  to refer to processors when subscripts are unnecessary. Each processor within the system has a unique identifier. The system is asynchronous in that no bound can be placed on the time required for a computation or for communication of a message. Processors have access to local clocks, but these clocks are not synchronized.

Each processor multicasts messages to the other processors. A processor receives all of its own multicast messages. Imposed on the communication medium is a logical ring with a token that controls the multicasting of messages. A *configuration* is the view of the system provided to the application, and a *ring* is the underlying mechanism on which the message ordering protocol operates. The *membership* of a configuration consists of a set of processor identifiers. Each ring has a *representative*, chosen deterministically from the membership, and an identifier that consists of a

ring sequence number and the identifier of the representative. A *regular* configuration has the same membership and identifier as its corresponding ring. A *transitional* configuration has a membership consisting of processors that are transitioning together from an old ring to a new ring.

We distinguish between the terms *generate*, *originate*, *receive* and *deliver*. A message is *generated* by the application for multicast on the communication medium, and is *originated* by the processor that multicasts it for the first time. A processor *receives* multicast messages, and *delivers* messages in total order to the application. Communication between processors is unreliable, and thus messages may need to be retransmitted and may be arbitrarily delayed, but the network is assumed not to partition. Communication channels are not assumed to be FIFO or authenticated.

We employ a public key cryptosystem such as RSA [17] in which each processor possesses a private key known only to itself with which it can digitally sign messages. Each processor is able to obtain the public keys of other processors to verify signed messages. The protocols also employ a message digest function such as MD5 [16] in which an arbitrary length input message  $m$  is mapped to a fixed length output  $d(m)$ . A processor signs a message  $m$  by using its private key to encrypt the digest  $d(m)$  of the message and by including this encrypted digest in the transmitted message.

The system is subject to processor faults. Processors are either correct or faulty. *Correct* processors always behave according to their specification. *Faulty* processors exhibit arbitrary (*Byzantine*) behavior. Because a Byzantine processor may behave as if it crashed, we include crash faults among the Byzantine faults we consider. For each regular and transitional configuration of size  $n$  we require that at least  $\lceil(2n + 1)/3\rceil$  processors are correct. Thus, the number  $k$  of Byzantine processors must satisfy  $k \leq \lfloor(n - 1)/3\rfloor$ .

## 3. Protocol Properties

The SecureRing protocols consist of a message delivery protocol, a primary component membership protocol, an unreliable Byzantine fault detector, and a message diffusion protocol. The organization is shown in Figure 1. Pseudocode for the protocols and proofs that the protocols satisfy the properties given below can be found in the full version of the paper [6].

### 3.1. Message Delivery Protocol

The message delivery protocol delivers two types of messages to the application. Regular messages are generated by the application for delivery to the application. Each regular message contains the identifier of the processor that originated it. A Configuration Change message informs the

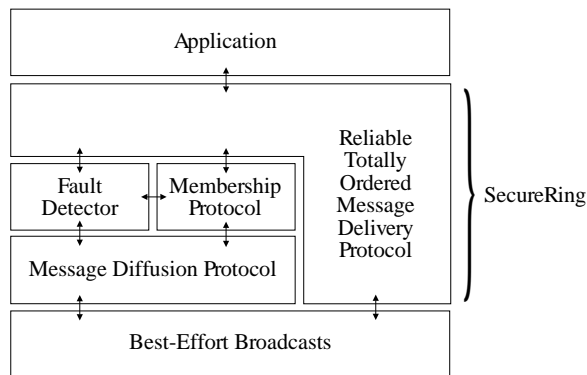


Figure 1: The SecureRing protocol stack.

application of a membership change, and is delivered in the message sequence along with the regular messages.

SecureRing provides *secure delivery* for each configuration  $C$ , which is defined in terms of the following properties:

- **Integrity.** For any message  $m$  that contains the identifier of processor  $p$ , every correct processor  $q$  delivers  $m$  at most once and, if  $p$  is correct, only if  $m$  was originated by  $p$ .
- **Uniqueness.** If a correct processor  $p$  delivers a message  $m$ , then no correct processor  $q$  delivers a mutant message  $m'$  having the same identifier as  $m$  but different contents.
- **Self-Delivery.** If a correct processor  $p$  originates a message  $m$  in configuration  $C$  and no configuration change occurs, then  $p$  delivers  $m$  in  $C$ . If a correct processor  $p$  installs consecutive configurations  $C_1$  and  $C_2$ , and  $p$  originates a message  $m$  in  $C_1$ , then  $p$  delivers  $m$  in  $C_1$  or in  $C_2$ .
- **Atomicity.** If correct processors  $p$  and  $q$  are both members of configuration  $C$  and no configuration change occurs, then  $p$  delivers a message  $m$  in  $C$  if and only if  $q$  delivers  $m$  in  $C$ . If correct processors  $p$  and  $q$  both install consecutive configurations  $C_1$  and  $C_2$ , then  $p$  delivers a message  $m$  in  $C_1$  if and only if  $q$  delivers  $m$  in  $C_1$ .
- **Total Order.** If correct processors  $p$  and  $q$  both deliver messages  $m_1$  and  $m_2$ , then  $p$  delivers  $m_1$  before  $m_2$  if and only if  $q$  delivers  $m_1$  before  $m_2$ .

### 3.2. Membership Protocol

The message delivery protocol depends on a membership protocol to handle some types of processor faults, while masking the effects of other types of processor faults. The faults handled by the membership protocol are called *detectable Byzantine faults*. We defer the discussion of

different types of faults to Section 6, where we also discuss unreliable fault detectors. The membership protocol receives information about detectable Byzantine faults from the fault detector and reconfigures the system to form a new membership.

To install a new configuration, two Configuration Change messages are delivered to the application. The first Configuration Change message introduces a transitional configuration that includes processors that are transitioning together from the old ring to the new ring. Some messages from the old configuration may be delivered in the transitional configuration if the requirements for secure delivery cannot be met in the old configuration but can be met in the smaller transitional configuration. The second Configuration Change message introduces the new regular configuration.

The membership protocol satisfies the following properties:

- **Agreement.** If a correct processor  $p$  installs a configuration  $C$ , then no correct processor  $q$  installs a configuration  $C'$  having the same identifier as  $C$  but a different membership.
- **Self-Inclusion.** If a correct processor  $p$  installs a configuration  $C$ , then  $C$  includes  $p$ .
- **Total Order.** If correct processors  $p$  and  $q$  both install configurations  $C_1$  and  $C_2$ , then  $p$  installs  $C_2$  after  $C_1$  if and only if  $q$  installs  $C_2$  after  $C_1$ .
- **Eventual Exclusion.** If  $p$  is a correct processor and  $q$  is a processor that has exhibited a detectable Byzantine fault, then there is a time after which  $p$  installs a configuration that excludes  $q$ , and  $p$  never subsequently installs a configuration that includes  $q$ .
- **Eventual Inclusion.** If  $p$  and  $q$  are correct processors, then there is a time after which  $p$  installs a configuration that includes  $q$ .

### 3.3. Fault Detector

The membership protocol depends on a fault detector with the following properties:

- **Eventual Strong Byzantine Completeness.** There is a time after which every processor that has exhibited a detectable Byzantine fault is permanently suspected by every correct processor.
- **Eventual Strong Accuracy.** There is a time after which every correct processor is never suspected by any correct processor.

While the accuracy property states that eventually every correct processor is *never* suspected by other correct processors, in practice it is sufficient that the condition must hold long enough for a new configuration to be installed.

### 3.4. Message Diffusion Protocol

The SecureRing protocols include a message diffusion protocol for the communication of special messages used by the membership protocol and fault detector, and for retransmission of messages during the recovery phase of the membership protocol prior to installing a new configuration. The message diffusion protocol is defined in terms of D-broadcast and D-receive primitives rather than in terms of origination and delivery. The primitives satisfy the following properties:

- **Self-Receipt.** If a correct processor D-broadcasts a message  $m$ , then it eventually D-receives  $m$ .
- **Uniform Receipt.** If a correct processor D-receives a message  $m$ , then every correct processor eventually D-receives  $m$ .

The diffusion protocol works as follows. When a processor receives a message for the first time, the processor relays it to all processors and then D-receives it. If communication is not reliable, the protocol does not guarantee the properties above and, indeed, no protocol can do so. However, in our implementation, if a correct processor does not D-receive a message, then that processor eventually comes to be suspected as faulty by other correct processors. Thus, a communication fault is viewed as a processor fault, and is considered in the resilience constraint.

The diffusion protocol has a message complexity of  $O(n^2)$ . To achieve high efficiency in fault-free operation, the SecureRing message delivery protocol employs a different strategy to achieve secure delivery during normal operation.

## 4. Message Delivery Protocol

SecureRing provides secure delivery of messages multicast by processors within the membership. The message delivery protocol provides the properties defined in Section 3.1 despite malicious attempts to disrupt it, provided that at least  $\lceil (2n + 1)/3 \rceil$  processors are correct and thus at most  $\lfloor (n - 1)/3 \rfloor$  processors are Byzantine in each regular and transitional configuration of size  $n$ .

### 4.1. The Data Structures

The message delivery protocol employs a number of message types and local data structures to provide secure delivery of messages. The proper forms of a regular message and a token are shown in Figure 2.

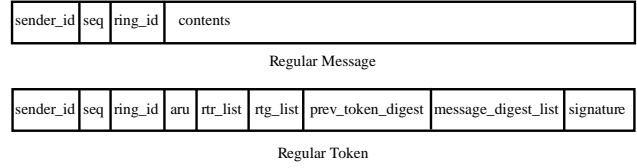


Figure 2: Proper forms of a regular message and a token.

**4.1.1. Regular Message.** Each regular message contains a number of fields, including the identifier of the processor originating the message, the ring identifier, the message sequence number, and the application-specific contents of the message. Regular messages are not signed. The identifier of a regular message consists of the *seq* and *ring\_id* fields.

**4.1.2. Regular Token.** To originate a regular message on the ring, a processor must hold the regular token, also referred to as the token. The sequence number of the message is obtained from the *seq* field of the token, which is incremented for each regular message originated on the ring. The *seq* field is also incremented for each multicast of the token; thus, each token contains a unique sequence number. All tokens are signed by the token holder before multicasting. Tokens are not delivered to the application, but they may be retransmitted. The token also contains the ring identifier in the *ring\_id* field. The identifier of a token consists of the *seq* and *ring\_id* fields.

The token also contains the *aru* (all received up to) field, which is a message identifier used to indicate that the processor multicasting the token has received each message with identifier less than or equal to the *aru*. The *rtr\_list* (retransmission request list) field is used to request the retransmission of any regular messages or tokens that the token holder has not received, and contains the identifiers of such messages. The *rtg\_list* (retransmission grant list) contains identifiers of messages or tokens that the token holder retransmitted. The digest of the token received by the token holder, *i.e.*, the previous token, is contained in the *previous\_token\_digest* field, and the *message\_digest\_list* field contains digests of the regular messages originated by the token holder.

We define the  $\succ_c$  relation for regular messages and tokens originated in configuration  $C$  as follows. If  $m$  is a regular message originated by a processor  $p$ ,  $t$  is the subsequent token multicast by  $p$ , and  $t$  contains the digest of  $m$  in the *message\_digest\_list* field, then we write  $t \succ_c m$ . If  $p_1$  and  $p_2$  are processors such that  $p_1$  is the immediate predecessor of  $p_2$  on the ring,  $t_1$  and  $t_2$  are successive tokens multicast by  $p_1$  and  $p_2$ , respectively, and  $t_2$  contains the digest of  $t_1$  in the *previous\_token\_digest* field, then we write  $t_2 \succ_c t_1$ . The  $\succ_c^*$  relation for regular messages and tokens is the transitive closure of the  $\succ_c$  relation.

## 4.2. The Protocol

Processors pack small messages generated by the application into larger messages (packets) that they sign and multicast. Packing improves the performance [13] by reducing the number of messages that must be multicast. If a processor has more messages to multicast than will fit into one packet, it follows the same procedure to form and multicast subsequent packets. From now on, we refer to a packet simply as a regular message.

**4.2.1. Token Processing.** When a processor  $p$  receives regular messages  $m_i$  and a subsequent token  $t$  from any other processor,  $p$  checks that the messages correspond to the *message\_digest\_list* field of the token, *i.e.*, for each message  $m_i$ ,  $t \succ_c m_i$ . If a message does not correspond to a digest in the subsequent token, it is ignored. Any token received without a valid signature is ignored. If  $p$  receives a token  $t_i$  from another processor  $q$  such that  $t_i \succ_c t_{i-1}$ , then  $p$  ignores any other token  $t_j$  from  $q$  such that  $t_j \succ_c t_{i-1}$ , *i.e.*,  $p$  accepts only one token from  $q$  that contains the digest of a given token in the *previous\_token\_digest* field.

When  $p$  becomes the token holder by receiving the token from its predecessor  $q$ ,  $p$  computes the digest of the token it received from  $q$  and places it into the *previous\_token\_digest* field of the token it will multicast. Processor  $p$  also updates the *aru* field to reflect the regular messages and tokens it has received. If  $p$  has failed to receive any regular messages or tokens that were previously originated by another processor (which is detected by a gap in the message sequence numbers),  $p$  places the identifiers of messages or tokens it is missing in the *rtr\_list* field of the token. If  $p$  has received a token  $t_1$  and a later token  $t_2$  such that  $t_2 \not\succeq_c^* t_1$ ,  $p$  does not acknowledge the receipt of  $t_2$  in the *aru* field of the token that it multicasts.

Processor  $p$  then retransmits messages and tokens it has received for which retransmission has been requested according to the following rules. Processor  $p$  retransmits any message  $m$  or token  $t$  whose identifier appeared in the *rtr\_list* field of any token it received, unless it has received subsequent tokens from  $k + 1$  other processors that contain the identifier of  $m$  or  $t$  in the *rtg\_list* field, where  $k$  is the resilience to Byzantine faults. For each message or token that  $p$  retransmits, it places the identifier in the *rtg\_list* field of the token. After retransmissions,  $p$  originates new regular messages generated by the application. For each new message that  $p$  originates, it places the digest of the message into the *message\_digest\_list* field of the token and increments the *seq* field of the token. When  $p$  has finished multicasting messages, it increments the *seq* field one more time and then signs and multicasts the token.

**4.2.2. Message Delivery.** A processor delivers a regular message  $m_1$  originated in a configuration  $C$  of size  $n$  as

follows. Let  $p_1, p_2, \dots, p_n$  denote the processors on the ring in the order of token rotation, let  $p_1$  originate message  $m_1$ , and let  $t_i$  be the first token multicast by processor  $p_i$  following the multicast of message  $m_1$ , *i.e.*,  $p_1$  multicasts  $t_1$  after multicasting  $m_1$ , subsequently  $p_2$  multicasts  $t_2$ , *etc.* A processor can deliver message  $m_1$  when (1) it has delivered every message  $m$  originated in  $C$  with sequence number less than that of  $m_1$  such that  $t_1 \succ_c^* m$ , (2) it has received token  $t_{k+1}$ , *i.e.*, the  $(k + 1)$ st token after  $m_1$ , where  $t_{k+1} \succ_c^* m_1$  and  $k$  is the resilience to Byzantine faults, and (3) it has received every regular message  $m$  and token  $t$  originated in  $C$  with sequence number less than that of token  $t_{k+1}$  such that  $t_{k+1} \succ_c^* t$  or  $t_{k+1} \succ_c^* m$ .

Secure delivery ensures that, if any correct processor  $p$  delivers a regular message  $m$ , then no correct processor delivers a mutant message  $m'$ . The reason is that (1) the  $k + 1$  successive tokens after the multicast of  $m$  that  $p$  received include at least one token  $t_i$  that was multicast by a correct processor  $p_i$ , and thus no processor receives a mutant token  $t'_i$  multicast by  $p_i$ , (2) a correct processor multicasts a token  $t_i$  such that  $t_i \succ_c t_{i-1}$  only when it has accepted token  $t_{i-1}$  from its predecessor, and (3) a correct processor accepts only one token  $t_{i-1}$  from its predecessor such that  $t_{i-1} \succ_c t_{i-2}$ .

## 5. Membership Protocol

The SecureRing protocols include a membership protocol that reconfigures the system when one or more processors exhibit a detectable Byzantine fault. The membership protocol forms a new ring consisting of apparently correct processors that are able to communicate with each other. The membership protocol satisfies the properties defined in Section 3.2 despite malicious attempts to disrupt it, provided that at least  $\lfloor (2n + 1)/3 \rfloor$  processors are correct and thus at most  $\lfloor (n - 1)/3 \rfloor$  processors are Byzantine in each regular and transitional configuration of size  $n$ . All communication in the membership protocol is performed using the message diffusion protocol described in Section 3.4.

### 5.1. The Data Structures

The membership protocol employs several special message types and local data structures. The proper forms of a Join message and a Commit token are shown in Figure 3.

**5.1.1. Join Message.** Processors attempt to reach agreement on a new membership by exchanging Join messages. Each Join message contains a number of fields, including *sender\_id*, *ring\_id*, *seq* which is used to order the Join messages D-broadcast by one sender, *proc\_set* which contains the set of processors that the sender is considering for membership in a new ring, and *fault\_set* which contains the

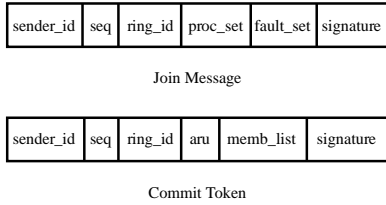


Figure 3: Proper forms of a Join message and a Commit token.

set of processors that the sender’s fault detector suspects of being faulty. Each Join message is signed. Join messages are not delivered to the application.

**5.1.2. Commit Token.** The Commit token is used to confirm that each processor is committed to the new membership once agreement has been reached. The Commit token contains a number of fields, including *sender\_id*, *ring\_id*, *seq* which is used to order the Commit tokens D-broadcast by one sender, and *memb\_list* which is a list of the identifiers of members of the new configuration. The Commit token also contains information necessary for the proper handling of messages from the old configuration that have not yet been delivered. For simplicity, this information is not shown in Figure 3. Each Commit token is signed.

**5.1.3. Local Variables.** Each processor maintains several local variables, including *my\_memb* which contains the membership of the processor’s current ring, and *my\_proc\_set* which contains the set of processors that the processor is considering for membership of a new ring. The local variable *my\_fault\_set* contains the set of processors that the sender’s fault detector suspects of being faulty. The local structure *my\_agreement* indicates whether each processor in the membership is in agreement with the processor’s *my\_proc\_set* and *my\_fault\_set*.

## 5.2. The Protocol

The SecureRing membership protocol can be described by a finite state machine with four states as shown in Figure 4.

**5.2.1. The Operational State.** The message delivery protocol is executed by a processor in the Operational state. When *my\_fault\_set* from the local fault detector module changes, the processor shifts to the Gather state. When the processor D-receives a properly signed foreign message from a processor that is not in *my\_memb* or *my\_fault\_set*, the processor updates *my\_proc\_set* so that it contains the union of *my\_memb* and the identifier of the foreign processor and shifts to the Gather state.

**5.2.2. The Gather State.** The message delivery protocol is executed by a processor in the Gather state for as long

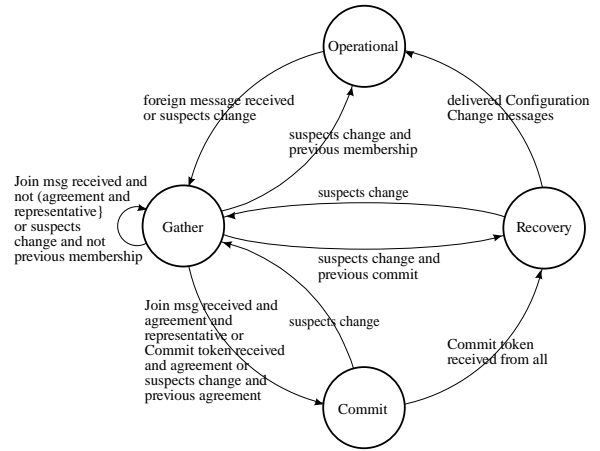


Figure 4: The finite state machine for the membership protocol.

as the token continues to circulate around the ring. In the Gather state, processors attempt to achieve agreement on the new membership. This is done by exchanging Join messages. When a processor shifts to the Gather state, it D-broadcasts a Join message in which *proc\_set* contains the processors in *my\_proc\_set* and *fault\_set* contains the processors in *my\_fault\_set* received as output from the fault detector. A processor periodically D-broadcasts its most recent Join message while it remains in the Gather state.

When a processor D-receives a Join message in the Gather state, it checks that the sequence number is greater than that of any previous Join message that the sender D-broadcast in the current configuration. Then, the processor compares the *proc\_set* and *fault\_set* fields with its local data structures *my\_proc\_set* and *my\_fault\_set*, respectively. If they are identical, the processor records the sender of the Join message as reaching agreement in its *my\_agreement* array. A processor reaches agreement when (1) it has recorded in its *my\_agreement* array every processor in the difference  $my\_proc\_set - my\_fault\_set$  as reaching agreement, and (2) the difference  $my\_proc\_set - my\_fault\_set$  contains at least  $\lceil (2n + 1)/3 \rceil$  of the processors from the local variable *my\_memb*, where *n* is the number of processors in *my\_memb*.

When a processor shifts to the Gather state, it retains information about the state from which it is transitioning and the values of *my\_proc\_set* and *my\_fault\_set* prior to the change that caused it to shift to the Gather state. When *my\_fault\_set* or *my\_proc\_set* changes while the processor is in the Gather state, the processor D-broadcasts a new Join message and clears its *my\_agreement* array. If both *my\_proc\_set* and *my\_fault\_set* are now identical to the previous values before the transition occurred, the processor shifts back to the state that it was in prior to the Gather state. Otherwise, the processor remains in the Gather state.

**5.2.3. The Commit State.** Once agreement is reached the representative, which is the member with the lowest identifier, generates a Commit token that contains a list of the members of the new ring. The Commit token is D-broadcast successively by each processor in ring order and allows for the exchange of information necessary to ensure proper handling of messages from the old ring. A processor periodically D-broadcasts its Commit token while it remains in the Commit state. If *my\_fault\_set* from the local fault detector module changes while a processor is in the Commit state, the processor D-broadcasts a new Join message and shifts to the Gather state.

**5.2.4. The Recovery State.** Once a processor has D-received a Commit token from each member of the new ring, it shifts to the Recovery state. When the representative shifts to the Recovery state, it generates and D-broadcasts a regular token. When a processor in the Recovery state becomes the token holder by D-receiving the token from its predecessor processor on the new ring, the processor D-broadcasts messages and tokens it has received but that had not been received by some of the processors transitioning from the old ring to the new ring. The processor then signs and D-broadcasts the token. No new messages are originated by a processor in the Recovery state. In the Recovery state the *rtr\_list* and *rtg\_list* fields of the token are used in the same way as in the message delivery protocol. If *my\_fault\_set* from the processor's fault detector changes while the processor is in the Recovery state, the processor D-broadcasts a Join message and shifts to the Gather state.

Once each processor has determined which messages meet the delivery criteria for the old and transitional configurations, each processor does the following in one atomic action:

- Delivers to the application those messages that can be delivered in the old configuration.
- Delivers the first Configuration Change message, which initiates the transitional configuration consisting of processors that are transitioning together from the old ring to the new ring.
- Delivers messages that could not be delivered in the old configuration, but can be delivered in the smaller transitional configuration.
- Delivers a second Configuration Change message that initiates the new regular configuration.
- Shifts to the Operational state.

**5.2.5. Termination.** Termination of the membership protocol is ensured by the properties of the fault detector: there is a time after which every processor that has exhibited a detectable Byzantine fault is permanently suspected by every correct processor, and there is a time after which every correct processor is never suspected by any correct processor.

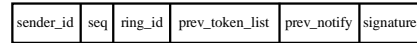


Figure 5: Proper form of a Notify message.

When these two properties hold, all of the correct processors reach agreement and install the new configuration. If the fault detector fails to satisfy eventual strong Byzantine completeness and eventual strong accuracy, the membership protocol may fail to satisfy the “liveness” properties of eventual exclusion and eventual inclusion. However, the membership protocol will continue to satisfy the “safety” properties of agreement, self-inclusion, and total order even if the fault detector properties are not satisfied.

## 6. Unreliable Byzantine Fault Detector

In the literature, fault detectors are defined in terms of abstract properties rather than in terms of a specific implementation. However, the question of how to support such an abstraction in a real system must be addressed. In a completely asynchronous system, it is impossible to implement a fault detector with the properties of eventual strong Byzantine completeness and eventual strong accuracy. However, many practical systems can be expected to exhibit reasonable behavior most of the time. If we assume that there are periods of stability with bounds on relative processor speeds and on message transmission times, and if we assume that signatures are unforgeable, then a Byzantine fault detector can be implemented.

The output of the fault detector consists of a list of processors that are suspected of exhibiting a detectable Byzantine fault. The particular instances of these faults are

- Sending mutant signed messages
- Sending a signed message that is not properly formed
- Failing forever to acknowledge the receipt of a regular message or a token
- Failing forever to send a message that is required by the message delivery protocol or the membership protocol.

### 6.1. The Data Structures

The fault detector uses a special message type called a Notify message to exchange information necessary for the detection of a Byzantine processor that has sent mutant tokens. Notify messages are communicated using the message diffusion protocol described in Section 3.4. Each Notify message contains *sender\_id*, *ring\_id*, *seq* which is used to order the Notify messages sent by one sender, and a list containing some tokens the sender has received as described below. Each Notify message is signed. The proper form of a Notify message is shown in Figure 5.

## 6.2. The Protocol

The fault detector monitors the messages sent by the message delivery and membership protocols, and provides its output to the membership protocol in the form of a list *my\_fault\_set* that contains the processors currently suspected by the local fault detector module.

**6.2.1. Mutant Messages.** When a processor  $p$  receives a token  $t_i$  from any other processor  $q$ ,  $p$  checks that  $t_i \succ_c t_{i-1}$  i.e., the *previous\_token\_digest* field of  $t_i$  corresponds to the previous token  $t_{i-1}$  and, if not,  $p$  D-broadcasts a Notify message. Each Notify message contains the  $k + 1$  most recent signed tokens  $p$  has received up to and including  $t_i$  and  $t_{i-1}$ , where  $k$  is the resilience to Byzantine faults. When a processor that has not D-broadcast a Notify message in the current configuration D-receives a properly formed Notify message, it D-broadcasts its own Notify message that contains the signed Notify message it D-received in the *prev\_notify* field. A processor that has D-broadcast a Notify message periodically D-broadcasts the same Notify message until it reaches agreement on a new membership. When a processor D-receives a Notify message it determines which processors have sent mutant tokens by using the list of signed tokens contained in the Notify message. It adds such processors to *my\_fault\_set*.

When a processor  $p$  D-receives a Join message that was D-broadcast by another processor  $q$ ,  $p$  checks if it has D-received a mutant Join message from  $q$  having the same identifier but different contents and, if so,  $p$  adds  $q$  to *my\_fault\_set*. Similarly, if  $p$  D-receives mutant Commit tokens or Notify messages sent by  $q$ ,  $p$  adds  $q$  to *my\_fault\_set*.

**6.2.2. Improperly Formed Messages.** If a processor receives a token, Join message, Commit token, or Notify message that is not of the proper form, the processor adds the sender to *my\_fault\_set*. The proper form of these messages is shown in Figures 2, 3 and 5 and is further described below.

A properly formed Notify message must contain either signed tokens  $t_i$  and  $t_{i-1}$  that are inconsistent such that  $t_i \not\succeq_c t_{i-1}$ , or another Notify message in the *prev\_notify* field that itself contains inconsistent tokens. If a processor D-receives a signed Notify message that is not properly formed, it adds the sender to *my\_fault\_set*.

When a processor  $p$  receives a token  $t_i$  from any other processor  $q$ ,  $p$  compares the fields of  $t_i$  with those of the token  $t_{i-1}$  that  $p$  received from  $q$ 's predecessor on the ring (the previous token holder). Processor  $p$  checks that the *seq* field of  $t_i$  differs from that of  $t_{i-1}$  by the number of message digests contained in  $t_i$  and, if not,  $p$  adds  $q$  to *my\_fault\_set*. Processor  $p$  also verifies that the *aru* field and the *rtr\_list* field in the token sent by  $q$  are mutually consistent and, if not,  $p$  adds  $q$  to *my\_fault\_set*.

The *aru* reported by a correct processor is monotonically non-decreasing with successive tokens. Therefore, when a processor  $p$  receives a regular or Commit token from another processor  $q$ ,  $p$  verifies that the *aru* reported is at least as great as the *aru* in the previous regular or Commit token sent by  $q$ ; if not,  $p$  adds  $q$  to *my\_fault\_set*.

A Byzantine processor could attempt to disrupt message ordering by advancing the *seq* field of the token, and including a corresponding message digest in the token, without actually multicasting a message. Without a mechanism to detect this type of fault, correct processors that fail to receive the non-existent message may be eliminated for failure to receive. Therefore, when a processor determines that at least  $\lceil (2n + 1)/3 \rceil$  processors in the membership have reported the same *aru* or included the same identifier in the *rtr\_list*, and that value has remained unchanged for a predetermined number of token rotations, the processor determines that the message is non-existent. In this event, the processor that sent the token containing the digest of the non-existent message is added to *my\_fault\_set*.

**6.2.3. Failure to Acknowledge Messages.** A processor cannot be allowed to fail to acknowledge a message indefinitely, because all of the other processors must buffer the missed message and all subsequent messages until the message is acknowledged. Therefore, a processor that multicasts a predetermined number of tokens in which the *aru* field is unchanged, or in which the same identifier appears in the *rtr\_list* field, is added to *my\_fault\_set*. A similar strategy is used for messages and tokens that are D-broadcast during the Recovery state.

**6.2.4. Failure to Send Messages.** The fault detector also makes use of timeouts in the detection of faults. The Token Loss timeout is set each time a processor receives a token and expires if it has not received the token from the next token holder in the allotted amount of time. The processor not heard from is presumed to be faulty, and is added to *my\_fault\_set*. The Token Loss timeout is also set when a processor shifts to the Recovery state and expires if the processor does not D-receive the token from the representative in the allotted amount of time. The representative is presumed to be faulty, and is added to *my\_fault\_set*. Additionally, the Token Loss timeout is set when a processor reaches agreement and when it D-receives a Commit token, and expires if the next Commit token is not D-received. The processor from which the Commit token was expected is presumed to be faulty, and is added to *my\_fault\_set*.

A Byzantine processor  $p$  that is the token holder could selectively multicast the token to some processors but not to its successor  $q$  on the ring. If this occurs, some processors that incur a Token Loss timeout may come to suspect  $q$  rather than  $p$ . Token retransmission has been implemented

to provide protection from this type of attack, and also to reduce the probability that the token is lost. Each time a processor  $p$  receives or sends a token  $t_i$ , it sets the Token Retransmission timeout. If  $p$  receives the next token  $t_{i+1}$ ,  $p$  resets the Token Retransmission timeout. If the Token Retransmission timeout expires,  $p$  forwards the token  $t_i$  that it last received or sent. The Token Retransmission timeout is shorter than the Token Loss timeout.

When a processor initially D-broadcasts a Notify message, it sets the Notify timeout. If the Notify timeout expires before a processor has D-received Notify messages from all of the members that are in the difference  $my\_memb - my\_fault\_set$ , the processor presumes that the processors from which it has not received a Notify message are faulty and adds them to  $my\_fault\_set$ .

The Agreement timeout is set when a processor in the Gather state has reached agreement with at least  $\lceil (2n + 1)/3 \rceil$  processors (where  $n$  is the number of processors in  $my\_memb$ ), but has not yet reached agreement with all of the processors in the difference  $my\_proc\_set - my\_fault\_set$ . The Agreement timeout is canceled if  $my\_fault\_set$  changes. If the Agreement timeout expires before agreement is reached, the processor presumes that the processors not reaching agreement are faulty and adds them to  $my\_fault\_set$ .

**6.2.5. Removal from the Fault Set.** When a processor  $q$  is added to  $my\_fault\_set$  of a processor  $p$  because a timeout expired, the entry in  $my\_fault\_set$  is appended to indicate that the suspicion is due to a timeout. If  $p$  subsequently receives the message it was expecting from  $q$ , it removes  $q$  from  $my\_fault\_set$ .

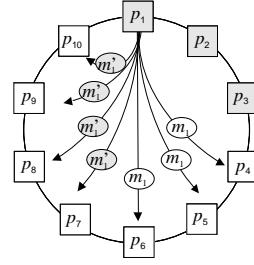
When a processor  $q$  is added to  $my\_fault\_set$  of a processor  $p$  because  $q$  has repeatedly failed to acknowledge a message or token, the entry in  $my\_fault\_set$  is appended to indicate that the suspicion is due to failure to acknowledge. If  $q$  subsequently acknowledges the message or token,  $p$  removes  $q$  from  $my\_fault\_set$ .

Processors that have sent an improperly formed message or mutant messages are never removed from  $my\_fault\_set$ , and are never readmitted to the membership.

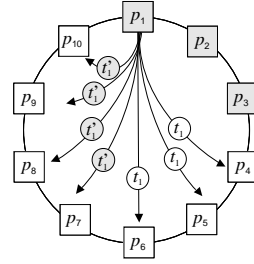
### 6.3. Example

We now give a possible scenario involving three Byzantine processors in a ring of ten processors. The example illustrates how multiple Byzantine processors could cooperate in sending mutant tokens. However, the fault detector is able to detect such faults.

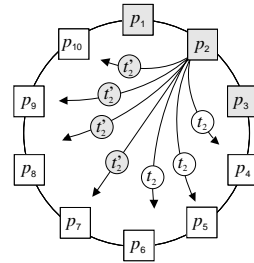
In Figure 6a, Byzantine processor  $p_1$  sends message  $m_1$  to correct processors  $p_4, p_5$  and  $p_6$ , but sends mutant message  $m'_1$  to correct processors  $p_7, p_8, p_9$  and  $p_{10}$ . To cover up, in Figure 6b,  $p_1$  sends token  $t_1$ , which contains



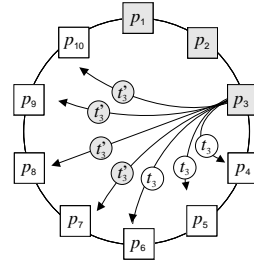
(a) Byzantine processor  $p_1$  sends mutant messages  $m_1$  and  $m'_1$ .



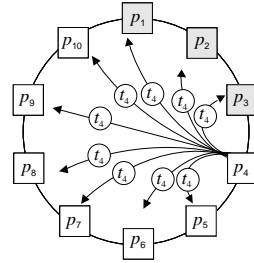
(b) Byzantine processor  $p_1$  sends mutant tokens  $t_1$  and  $t'_1$ .



(c) Byzantine processor  $p_2$  sends mutant tokens  $t_2$  and  $t'_2$ .



(d) Byzantine processor  $p_3$  sends mutant tokens  $t_3$  and  $t'_3$ .



(e) Correct processor  $p_4$  sends mutant tokens  $t_4$  and  $t'_4$ .

Figure 6: An example in which multiple Byzantine processors cooperate by sending mutant tokens. The Byzantine processors and mutant tokens are shaded.

a digest of  $m_1$ , to processors  $p_4, p_5$  and  $p_6$ , but sends  $t'_1$ , which contains a digest of  $m'_1$ , to processors  $p_7, p_8, p_9$  and  $p_{10}$ .

In Figure 6c, Byzantine processor  $p_2$  covers up for  $p_1$  by sending tokens  $t_2$  and  $t'_2$ , which contain digests of  $t_1$  and  $t'_1$ , respectively. Similarly, in Figure 6d, Byzantine processor  $p_3$  covers up for  $p_1$  and  $p_2$  by sending tokens  $t_3$  and  $t'_3$ , which contain digests of  $t_2$  and  $t'_2$ , respectively.

As shown in Figure 6e, correct processor  $p_4$  sends  $t_4$ , which contains a digest of  $t_3$ , to all processors. Subsequently, the fault detector modules at processors  $p_7, p_8, p_9$  and  $p_{10}$  detect the inconsistency between  $t'_3$  and the digest of  $t_3$  that is contained in  $t_4$  and multicast Notify messages. This leads to the removal of processors  $p_1, p_2$  and  $p_3$  from the membership. Additionally, processors  $p_7, p_8, p_9$  and  $p_{10}$  will not deliver message  $m'_1$ .

## 7. Conclusion

We have presented the SecureRing protocols, which provide reliable ordered message delivery and group membership services within an asynchronous distributed system using a logical ring and a token that contains ordering and fault detection information. The protocols are resistant to Byzantine faults. The message delivery protocol ensures that all correct processors within a configuration deliver the same messages in the same total order, and achieves this without using multiple rounds of message exchange or requiring digital signatures on every message. The membership protocol ensures that correct members agree on changes to the membership, that correct processors are eventually included in the membership, and that processors that exhibit detectable Byzantine faults are eventually excluded from the membership. To provide these message delivery and group membership services, the protocols make use of an unreliable Byzantine fault detector.

## Acknowledgments

We wish to thank the anonymous referees for their constructive comments and also Michael Reiter for helpful discussions.

## References

- [1] D. A. Agarwal, *Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks*. PhD Dissertation. Department of Electrical and Computer Engineering. University of California, Santa Barbara (August 1994).
- [2] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal and P. Ciarfella, "The Totem single-ring ordering and membership protocol," *ACM Transactions on Computer Systems*, vol. 13, no. 4 (November 1995), pp. 311-342.
- [3] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2 (March 1996), pp. 225-267.
- [4] M. J. Fischer, N. A. Lynch and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2 (April 1985), pp. 374-382.
- [5] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith, "Unreliable Byzantine fault detectors for solving consensus," Department of Electrical and Computer Engineering. University of California, Santa Barbara, Technical Report 97-14.
- [6] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith, "The SecureRing group communication protocols," Department of Electrical and Computer Engineering. University of California, Santa Barbara, Technical Report 97-26.
- [7] D. Malkhi and M. Reiter, "A high-throughput secure reliable multicast protocol," *Proceedings of the 9th Computer Security Foundations Workshop*, Kenmore, Ireland (June 1996), pp. 9-17.
- [8] D. Malkhi and M. Reiter, "Unreliable intrusion detection in distributed computations," *Proceedings of the 10th Computer Security Foundations Workshop*, Rockport, MA (June 1997) pp. 116-124.
- [9] P. M. Melliar-Smith, L. E. Moser and V. Agrawala, "Broadcast protocols for distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1 (January 1990), pp. 17-25.
- [10] L. E. Moser and P. M. Melliar-Smith, "Total ordering algorithms for asynchronous Byzantine systems," *Proceedings of WDAG '95, 9th International Workshop on Distributed Algorithms*, Le Mont Saint Michel, France (September 1995), Lecture Notes in Computer Science 972, pp. 242-256.
- [11] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia and C. A. Lingley-Papadopoulos, "Totem: A fault-tolerant multicast group communication system," *Communications of the ACM*, vol. 39, no. 4 (April 1996), pp. 54-63.
- [12] L. E. Moser, P. M. Melliar-Smith and V. Agrawala, "Processor membership in asynchronous distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 5 (May 1994), pp. 459-473.
- [13] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "Message packing as a performance enhancement strategy with application to the Totem protocols," *IEEE Global Telecommunications Conference*, London, UK (November 1996), pp. 649-653.
- [14] M. K. Reiter, "Secure agreement protocols: Reliable and atomic group multicast in Rampart," *Proceedings of the 2nd ACM Conference on Computer and Communications Security* (November 1994), pp. 68-80.
- [15] M. K. Reiter, "A secure group membership protocol," *IEEE Transactions on Software Engineering*, vol. 22, no. 1 (January 1996), pp. 31-42.
- [16] R. L. Rivest, "The MD5 message digest algorithm," *Internet Activities Board* (April 1992).
- [17] R. L. Rivest, A. Shamir and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2 (February 1978), pp. 120-126.