SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 27(7), 813–850 (JULY 1997)

Implementation of a Reliable Multicast Protocol

WEIJIA JIA
Department of Computer Science, City University of Hong Kong, 83 Tat Chee Avenue, Kowloon, Hong Kong
(e-mail: wjia@cs.cityu.edu.hk)

SUMMARY

A reliable multicast protocol (RMP) based on a logical token ring approach can achieve agreement of a group of operational processes in distributed systems. The structure of RMP is modulated into component protocols that achieve total ordering, atomicity of multicast messages; dynamic group configuration and fault-tolerance cooperatively. RMP uses a virtual token to order multicast messages in a logical process ring. It is highly efficient over networks and its algorithm design and implementation are presented. The use of a state machine approach simplifies this complex system implementation. Experience and lessons drawn from RMP and general techniques applied to group communication protocol are also described. @1997 by John Wiley & Sons, Ltd.

KEY WORDS: multicast protocol; fault tolerance; group communication; implementation; state machines

INTRODUCTION

Distributed systems providing overall services need careful design and implementation to preserve the consistency of shared information among the cooperative processes. To increase the availability of the systems, the key idea is to replicate system servers running on the distributed processors.[1] Achieving the correct function of processors requires the processes to have a consistent view of cooperative tasks in a processor/process group. However, the underlying distributed systems are imperfect, and they are subject to a number of possible faults, such as a processor crashes, and the unreliable communication medium may lose or reorder messages. The network may partition into separate segments. Reaching a consistent view or agreement is known to be impossible in a realistic asynchronous distributed system.[2] At the present, most practical systems use a reliable multicast protocol to achieve such a view among the operational processes. Reliable multicast, in contrast to unicast communication, which involves a single source and a single destination, refers to a single source and a set of destinations. In this paper, ‘process’ is used to denote a protocol server process for a site instead of an application process. We intend to describe a reliable multicast protocol that guarantees the following properties: (1) *total ordering*, a sequence of delivered messages is identical at all operational receivers; and (2) *atomicity*, a message issued by a sender either reaches all correct operational receivers in a group or none of them. In this paper, a ‘correct’ process

CCC 0038–0644/97/070813–38 $17.50
© 1997 by John Wiley & Sons, Ltd.
Received 4 June 1996
Revised 5 September and 25 October 1996

means that the process is operational; it executes the communication protocol correctly and follows the protocol specification.

To see the usefulness of the reliable multicast protocol, consider, for example, a well-known 'transaction commit problem', which arises in distributed database systems.[3] The problem is for all the data manager processes that have participated in the processing of a particular transaction to agree on whether to install the transaction's results in the database or discard them. Whatever decision is made, all data managers must make the same decision to preserve the consistency of the database. The protocol that can be used to implement such an atomic commit is usually called 'safe' or 'uniform' multicast. In this paper, *atomic* multicast is used as a synonym for *safe* or *uniform* multicast.

Reliable Multicast Protocol (RMP) operates in a single process logical token ring, as described in References 4 and 5. Logical token ring has been considered as a simple and efficient approach in the design of a multicast protocol, because the discrete multicast messages of many distributed processes are reduced to one process holding the token and multicasting a totally-ordered message at one time, and the process is called 'token holder' or 'token process'. This method substantially simplifies the synchronous requirements of the application processes. Many existing multicast algorithms and protocols have been presented in the literature and have addressed issues of the implementation techniques.[6–14] In this paper, the characteristics of RMP are given, especially its implementation and application for distributed dynamic action system.[5]

Figure 1 illustrates the system structure, which consists of a group of sites, i.e. self-contained computers including high level application software. The sites are loosely coupled by a network, and each RMP server process, accepts multicast requests from its local application via a submittal queue *QSUB* and delivers messages to the application via a delivery queue *QCMT*. Note that the process interacts with its applications through a TCP connection. Once *QSUB* is full, the process stops accepting messages and the applications are blocked. As along as the process has made room for *QSUB*, it is able to receive the message request from its application,

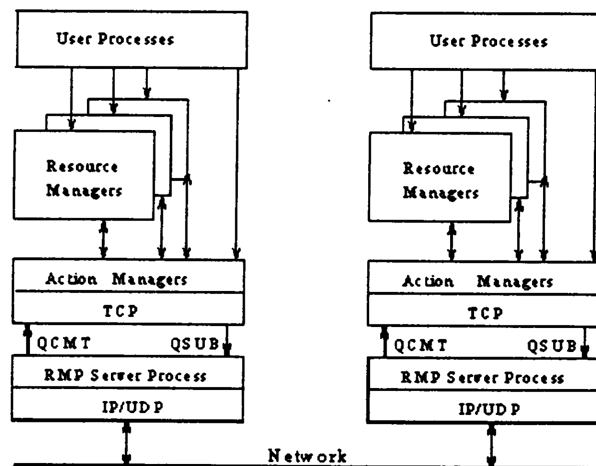

Figure 1. System structure

here the application is referred to as an action manager. Hence, it is assumed that *QSUB* does not overflow in this paper. The user application processes running on the upper level of the system access common resources (e.g. files, shared objects and database systems, etc.) and share the common information via resource and action manager systems. The action manager handles the transaction dependency and offers consistent commitment or abortion functions by communicating directly with the process. Assuming the continued execution of the processes in a group of sites, the action manager on one of the sites, in the case of needing reliable multicast service, opens a port to the process, which listens to the connection request and accepts the connection. Upon building the connection, the action manager is able to send a data message to the group of action managers via the RMP processes by entrusting the message to the port. On receiving a multicast message from an RMP process, the action manager can be sure that the message has been received in the same relative order by the action managers through RMP, even in the presence of communication faults, site crashes or network partitioning.

Modular structure of RMP

Although this paper presents an implementation, the techniques and algorithms described can be applied to general fault-tolerant multicast protocol design and implementation. A state machine approach is applied for the modular design and implementation of RMP. Therefore, RMP is taken as the modular composition of sub-protocols that work cooperatively in achieving message ordering, reliability and system fault-tolerance. This approach is similar to the microprotocol approach introduced in References 16–18. To better understand the structure of RMP, the modular hierarchy is shown in Figure 2. RMP is a composition of *total ordering*, *atomicity*, *fault-tolerant* and *membership* protocols that achieve services of efficiency and reliability for multicast messages. In RMP, the total order respects the *causal order* of Lamport.[19] Note that message `receive` is further distinguished from `delivery`. Message receive means that a message is received by a RMP process, while message delivery means that a message is committed to high level applications by the process.

The total ordering protocol guarantees the message multicast and delivery to applications in the same relative order. If a message desires a safe property (i.e. atomicity), an atomic protocol is invoked. During the normal message multicast, by *timeout*, if any fault is suspected, the fault-tolerant protocol is invoked to perform fault detection and location. Any membership change of the process group due to fault recovery or the dynamic group is conducted by the membership protocol to handle the member join/leave, merge of two partitioning segments, etc. The membership protocol forming a new group must guarantee membership change atomicity and have a consensus among the operational sites.

Application of RMP for dynamic action system

The dynamic action model[15] offers system support for highly dynamic process cooperation. Dynamic actions facilitate the task of writing distributed applications without adopting the restrictive computational model of database transactions. Like transactions in a distributed database, the action model achieves computation distribution and failure transparency for high level applications. Committing an action is

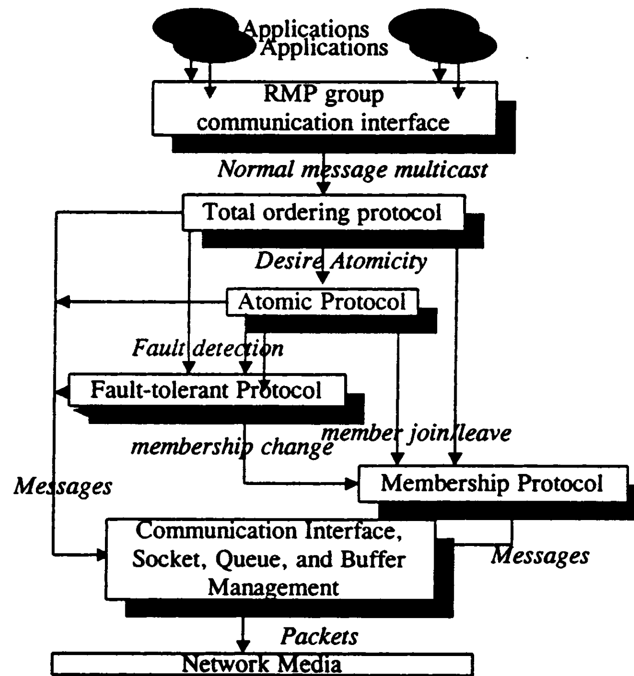

Figure 2. RMP hierarchy structure

accomplished by running a so-called *commit protocol.* A commit protocol guarantees that all sites involved in commit processing come to a consistent decision. It ensures that an action will either be committed by making its results permanent or will be aborted without affecting the system. In the well-known centralized two-phase commit protocol, the commit decision is determined by a centralized instance called the *coordinator.* A protocol enabling the surviving sites to come to a decision without waiting for the faulty site to recover is known as a *non-blocking* commit protocol.

Commit protocols avoiding a centralized coordinator are know as *decentralized* commit protocols. Within a decentralized commit protocol, each site involved represents an independent participant that can determine the commit decision based on the messages exchanged with all other participants. In the case of a site fault the faulty participant is able to determine the commit decision as long as at least one of the surviving participant knows the commit decision. Another advantage of a decentralized protocol is that the message announcing the outcome of a commit protocol is no longer needed, since the commit decision can be determined locally by the participant itself.

In the dynamic action model, the idea was to design a sophisticated commit protocol that resolved the blocking problem and exploited the advantages of decentralized communication within the underlying communication protocol represented by RMP. A site fault may occur at any time during the execution of a commit protocol. To avoid blocking situations, the following properties supplied by the RMP are exploited as message *atomicity*, *ordering* and *site fault detection.* It is important to note that RMP guarantees that the message announcing a site fault is inserted into the total ordering of all messages. Inserting the site fault message into the total

message ordering enables a participant of the commit protocol to make the following assumptions about the messages received by the surviving sites: at the time the site fault occurs, the participant of a non-faulty site has received all messages the participant of the faulty site received before it crashed. Furthermore, it is ensured that at the time the surviving participants recognize the site fault, they have received all messages in the same order.

On the other hand, the surviving sites have to agree on a decision about the outcome of the running commit protocol that is consistent with the view of the faulty site. This problem is not always trivial, especially if a message of the faulty site has not yet been received by other participants, it is not clear whether the participant is already in a consistent state. However, if the faulty site received all the messages from all other participants, the action may even be committed at the faulty site. Thus, additional information about the state of the faulty participant is needed, and the information is gathered in total message ordering.

To increase the system efficiency, several commit protocols are allowed to execute in parallel. This parallel execution may lead to a problem if actions are involved in different commit protocol executions called 'competitive commit requests'. To resolve this problem, a concurrently running commit request is delayed until the competitive commit request has been terminated. That is, the execution of competitive commit requests must be sequential. However, sequentializing commit requests may cause a deadlock. To avoid deadlocks, again, the total ordering property of the RMP is exploited. To do so, the executions of concurrently running commit protocols are coordinated so that different commit requests are performed by each participant in the same order. To achieve this order, the initiator of a commit protocol delays the commit processing until the commit request message is received by the initiator itself. At this moment, the initiator is sure that the commit request message has been received by all operational participates, i.e. the message is atomic, a property guaranteed by RMP.

ASSUMPTIONS AND DATA STRUCTURE

System environment

Currently, RMP is implemented based on the Unix 4.3BSD operating system, which provides a rich set of distributed program facilities that could easily be used in supporting resource sharing in a distributed environment.[20] The initial implementation envisages RMP using sockets on a local area network connected by 10-Mb/sec Ethernet. A socket is an abstract object from which messages are sent and received. Both connection and connectionless sockets have been used: between client processes and RMP processes, Unix domain sockets are used for passing connection byte streams whereas datagram sockets are used to transmit messages across a network that models potential unreliable, connectionless packet communication. The datagram socket id of each endpoint of communication is defined prior to the transmission of any data, and is maintained at each process so that it can be presented at any time.

Communication and timer assumptions

The strongest assumption that can be made about inter-process communication is that any message sent by a correct process to another correct process is always received within a given delay—the so-called *synchronous communication* assumption. The nice property of about this assumption is that one process can reliably detect whether another is alive just by sending it a query and waiting for a known bounded time for a response. Unfortunately, in a system where processes must communicate over a shared network, such perfection is guaranteed with only a certain probability, by using multiple communication paths and/or message retransmission. It is impossible to give a probabilistic guarantee, since the actual load on the network may be totally unpredictable.

The alternative approach is to consider that there is no known limit on the time it takes for a message to reach its destination. Protocols designed without knowledge of time limits could easily be ported from one environment to another, since they would operate correctly whatever the performance of the network. But with such *asynchronous communication*, a process cannot decide whether another process has crashed or whether its query or the expected response is still on its way across the network. In practice, it is essential to introduce some notion of time so that processes know how long to wait for an expected response before suspecting that the originator of the response might have failed.

For the implementation of RMP, the datagram communication protocol UDP has been used in an asynchronous networks (e.g. Ethernet), which provides a cheap means for any process to send a message to any other process. The datagram message packet may be lost, duplicated or out of order. The lost message must be retransmitted. Between the emission of a datagram message by a source process and the moment the message is received at its destination by the target process, there is an arbitrary random delay δ. Since the delay can be variable, for the sake of practical system implementation, the value of δ is considered as a constant such that a datagram message travelling more than δ time units is taken as being lost. Therefore, δ is taken as the worst case or an approximate measurement for the point-to-point message transmission over an asynchronous network. Similarly, among a group of processes receiving a datagram message multicast by a source, there is also a random delay Δ, such that a multicast datagram message that travels more than Δ time units between any two processes in the group is also taken as being lost. Such delays should be designated to prevent situations in which a process waits forever for a message from another process that will never arrive due to the failure of the process. Datagram delays are established by studying statistics about network behavior under various load patterns, so as to ensure that point-to-point/broadcast datagram message transmission delays are smaller than δ (Δ) with a very high probability.

Fault assumptions

In networks, typically in local area networks such as Ethernet or IBM token ring, each site is connected to a network interface. The interface monitors the network and copies messages identified, with its address code, into a buffer that can be accessed by a connected site. Unfortunately, there is no guarantee that a site will receive every message addressed to it. To model such a circumstance, the following faults are assumed:

(a) Messages may be lost because of a buffer overflow; they may be discarded, out of order or duplicated due to a transmission error. However, all messages received at a site are free of transmission errors.

(b) A site can fail or disconnect from the network. A failed site may suddenly halt, killing all the processes running there or experience timing faults. However, a failed site is therefore free of malicious actions, and is referred to as 'fail stop' model. A transient fault is also modeled as a fail stop. The transient fault happens to a site when it has no communication with other sites due to communication delay or heavy load. Our experience has suggested that operating systems, particularly Unix, are prone to pauses of a few seconds even if the site has not failed.[12] RMP implementation is made to accept such occasional pauses during which a part of the system may stop.

(c) Network may partition for reasons such as the failure of bridges, gateways, network switches and communication links. Network partition faults are accommodated by group reconfiguration. RMP continues to operate with the majority of members in a group. In a partitioned system, the processes in a segment of the partition appear to have failed to the processes in the other segments. In case that there is no majority partition available when the number of segments is greater than 2, RMP in the segments can be interrupted by users to allow one of the segments to continue its normal message multicast by changing the majority condition. This condition provides RMP with flexibility when it executes in a distributed environment.

Data structure

Let G be the process group and $G = \{a, b, c, \ldots\}$ the size $|G| = N$ and $N \geq 2$. A process logical ring defined as $R = (P_0, P_1, \ldots, P_{n-1})$ where $P_i \in R$ is a process notion corresponding to a specific process $x \in G$ and $n \leq N$. For simplicity, hereafter, P_{i} is used to denote x itself. Index $i \in \{0, \ldots, n-1\}$ is used for the position id of P_i on R. Note that the ids are dynamically attached to the processes with each change of R structure, e.g. assume an old ring R and and a new ring R', a process $P_i \in R$ is not necessarily the same process as $P_i \in R'$ even both R and R' contain the same set of processes. The differences between R and G are their logical structure and size. N is used for the group size and n for the size of R. R is said the current logical token ring if R is operational and $n > \lfloor N/2+1 \rfloor$ (majority consideration). In case $n=N$, it indicates that all processes in G are operational. Each process P_i maintains R, $QSUB_i$, $QCMT_i$ and the following entities:

(a) $QDAT_i$ buffers the totally-ordered multicast messages received at P_i but not yet delivered to the applications;

(b) $QSUS_i$: buffers the out of order multicast messages received from other processes on R;

(c) S_i, the system-wide sequence number for message multicast, initially 0;

(d) *Version*: the version of R, initially 0, increments with each change of R;

(e) K_i: an acknowledgment array with size n, (initially all 0), recording the sequence numbers of all the processes on the ring R locally seen by P_i.

Let Q be a queue and m be a message, Q:=Q+m denotes the enqueue operation on Q (append m at the end of Q) and Q:=Q−m dequeues m from Q. $Q[i]$ is a

message stored in the ith position in Q and $Q[0]=head(Q)$ is the first message in Q. In RMP, all the queues are implemented with linked lists and their space can be dynamically expended or decreased.

Message packet and control information

To implement the message packets, one must define their types. RMP denotes the messages as *upstream* and *downstream*. The *upstream* message that a process receives or delivers from/to the upper application is defined as an Information Data Unit (IDU), and a message it sends/receives to/from the network is defined as a Packet Data Unit (PDU). The process attaches a PDU header to the IDU and makes it a PDU for network transmission. Likewise, a PDU can be stripped from its PDU header to become an IDU for delivery. Their formats are given below:

(a) *IDU*: There are two types of IDU. One is the data IDU and the other the membership IDU. The data IDU is used for data message passing between applications and the processes. The membership IDU is delivered by the process to notify the applications of a membership change. The format of the membership IDU is shown in Figure 3(a), in which sid_i are ids of member processes (or member internet ids) in the group; variable vi is used for information of applications about whether the member is new (vi=0) or old member (vi>0) in G. RMP defines an **`IDU_HEADER`** for the use and information of high level applications. It consists of components (**`sender_id, port, length, flag`**) in which `sender_id` is the sender process id; **`port`** is used for communication with RMP process; `length` indicates the IDU in terms of byte length and `flag` is used by the applications to inform RMP if the message is an urgent message. For example, `flag =1` can be as the indication that the IDU requires atomicity. Actually, `flag` can be used to encoded multiple types of IDU, and the issue is out of scope of the paper.

(b) *PDU*: in general, RMP control information is included in a PDU header with the type **`PDU_HEADER`**. Its instance is attached to an IDU before being sent to a network, and is denoted as `header=(protocolID, version, sender, mid, s, k, code, len, coordinator)` where `protocolID` is the id of the RMP that is currently running; `version` is the version of R; `sender` is the

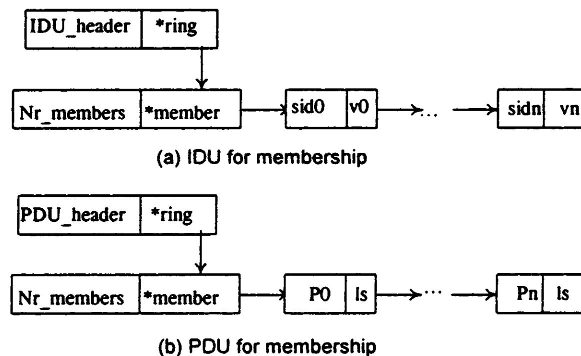

Figure 3. Packet for logical ring

site id (`sid`) of the PDU sender, i.e. the internet address; `mid = (sid, ls)` is a unique message id in which `sid` is the originator `sid` of the PDU;* *ls* is the local sequence number of the originator for this PDU; `s` is a sequence number denoting a total order of PDU if it is a multicast message; otherwise, it represents an acknowledgment of the `sender` about the previous messages received; `k` is called safe parameter, showing the sender's view of maximum order of the messages received by every member on the ring; `code` denotes the type of `m` and it is interpreted by RMP as external events (see `event` section below). `len` is the length of the PDU in bytes; `coordinator` is the `sid` of the ring coordinator. There are three kinds of PDUs: *control*, *data* and *ring*. A control PDU only has a `header`. A data PDU is a pair of `(header, *data)` where `*data` is the pointer to an IDU; a ring PDU is depicted in Figure 3(b), in which P_i is the member `sid` and `ls` is the local message sequence number of P_i.

PROTOCOL MODULE

In this section, the sub-protocol algorithms of RMP are described. Their integrated and cooperative implementations are presented in later sections. The key idea of guaranteeing the correct operation of RMP is that each process maintains a consistent logical ring R. During the ring construction, a process that first executes RMP is called the ring coordinator and it creates the ring, assigning each member a unique id P_i where $i \in \{0,\ldots,n-1\}$ that is a one-to-one mapping of a `sid`.

Ring creation

Initially, a process executing RMP only knows the number of its peer processes in G. A process learns its peer by reception of messages from its RMP port. Suppose process P first executes RMP. It creates a logical ring for a group of n processes. P serves as the ring coordinator by setting $R[0]=sid(P)$, proposing a ring version with 1 (`R.Version=1`), broadcasting a reformation message, inviting the processes to join R. The processes using RMP ports listen to the invitation message. If they agree with the invitation, a positive `ack` is sent to P and accept it as the coordinator. Suppose P has collected i members into R, when P' joins, P sets $R[i]=sid(P')$. P multicasts R to the members and the latter install R and enter into normal operation. During the invitation procedure, a process responds with positive or negative messages, depending on the versions of the sender and its own, as illustrated below. Should P ever receive a negative response, it terminates configuration and waits for further invitations. It is a common case that once a set of processes execute RMP initially, there is no predefined ring coordinator; a process seeing this fact may vote itself as the ring coordinator. A competition may occur if several processes compete for the coordinator. To avoid the competition to some extent, RMP uses three rules for the selection and authorization of a candidate. Note that the group coordinator can be differentiated from a coordinator candidate. A coordinator candidate is selected from between two processes, and it becomes the group coordinator if the majority of processes elect it as the candidate. The election rules are given below. Let

* In general `sender = mid.sid`. In case the PDU is a delegate message, the sender is different from the packet originator, then `sender ≠ mid.sid`.

processes P and P' compete with each other: (**R1**) If `P.Version>P'.Version`, P is chosen as the coordinator candidate by P'; (**R2**) If `P.Version=P'.Version` and `P.S>P'.S` (the sequence number), P' votes P as the coordinator candidate; (**R3**) if `P.Version=P'.Version` and `P.S=P'.S` and `sid(P)>sid(P')`, then P is elected as the coordinator candidate by P'.

We put a priority on the version of ring (**R1**) because a process that is not up-to-date cannot be the ring coordinator. (**R2**) takes the sequence number as the second priority factor if two processes on the same ring compete with each other. It is evident that the process with fewer messages should first complete its lost messages. As a result of applying (**R1**) through (**R3**), P' answers a positive message to P, and inversely, P sends a negative response to P'. Given the selection rules, when the protocol starts operation, each process waits a random time interval for an invitation. Through experiments we have found that using rules (**R1–R3**) will reduce the reformation time and avoid a race situation to some extent. After the election, the coordinator (P_0) constructs a logical ring based on the positive response members and delivers the ring across the group as described in the following subsections.

Ordered protocol

The value of a total order for a multicast message in R is recorded in a sequence number S. There is a virtual token rotating in R. At one time, there is only one member holding the token, multicasting a total ordered message, and incrementing S by 1. The token implicitly circulates to the next process on R. Each member uses S for checking the message total ordering it received. All members expect the totally-ordered incoming message from the token holder. Each member predicts the id of the token holder by the value of S plus 1 modulo the size of R, that is denoted as $S \oplus 1$. A member wishing to multicast data messages can wait until holding the token. If it has an urgent message, it can send the message to the current token holder, requesting the holder to multicast on behalf of it. If the token holder does not have data message to send, it has to multicast a *NULL* message to let the token go to the next process on R, as shown in Figure 4.

It may be that different members have inconsistent values of S, which indicates that members with fewer values of S may lose some ordered multicast messages.

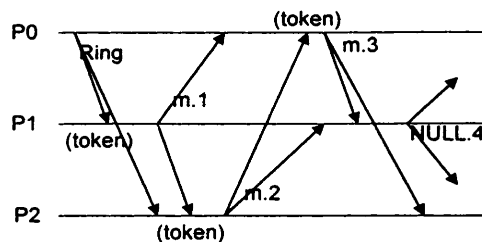

Figure 4. A group constitutes three members. P_0 *multicasts a ring to all the processes, informs the membership and passes the token to* P_1. P_1 *then transmits* m_1, *piggybacking the token to* P_2 *implicitly, and* P_2 *multicasts* m.*2. On receiving* m.*1, the receivers advance their* S *by* S:=S+*1 and know that* P_2 *is the second token holder, and so forth. If* P_{n-1} *has multicasted the* (n−1)*th message, the token transfers to* P_0 *again for multicast of the* n*th message. If the token holder has nothing to send, a* **NULL** *message is transmitted*

The member, upon reception of an ordered multicast message, can detect the lost messages via the gap of its *S* and the incoming message order. As a result, retransmission of the lost messages are immediately carried out without further delay. In RMP, a receiver initiates the message retransmission. Assume that *P* received *m* and detects the missed messages via $S<m.s$, it sends a request to `m.sender`, asking for retransmission of messages with an order between $m.s-S$. On the other hand, if $S>m.s$, *P* retransmits the messages with order in $S-m.s$ to `m.sender`. Intuitively, using *S* to monitor the token holder seems to cause problem because of inconsistent values of *S*. In fact, this scheme helps individual members to monitor the expected incoming message efficiently.

Multicasting a message from a user application to the network until it is received by the group members requires steps of message picketing and sending to network. A user application on top of P_i calls the RMP interface function `rmp_multicast(data)` that generates an IDU header, and inserts the IDU into the RMP port by a Unix system call `write(rmp_port, IDU, idu_length)`. Figure 5 depicts the steps taken for a data message IDU received by RMP and multicast to the network. In the diagram, RMP uses the Unix call `recv` to receive the IDU. The IDU is converted into a PDU by adding a PDU header with order *S*, and it is queued in *QSUB* for multicast. On holding the token, P_i multicasts the PDU to the network; if not, the flag of the IDU is checked to see if it is a `fast` type. In the latter case, the PDU is forwarded to the current token holder anticipated by P_i, which is P_j, where $j=S_i+1$.

Reception of a multicast message *m* requires P_i to save the sender information in the corresponding membership, say $R[j]$, where `m.sender`$=P_j$. The information includes the sender local sequence number. *m.s* is checked to see if it is in the expected message order, i.e. if $m.s=S_i+1$. If so, *m* is queued in *QDAT* for delivery. Otherwise, *m* is queued in *QSUS* until it is in the total order in terms of P_i and it is linked to *QDAT*. If the flag of *m* requires atomic delivery, the atomic protocol is invoked as described in the next subsection.

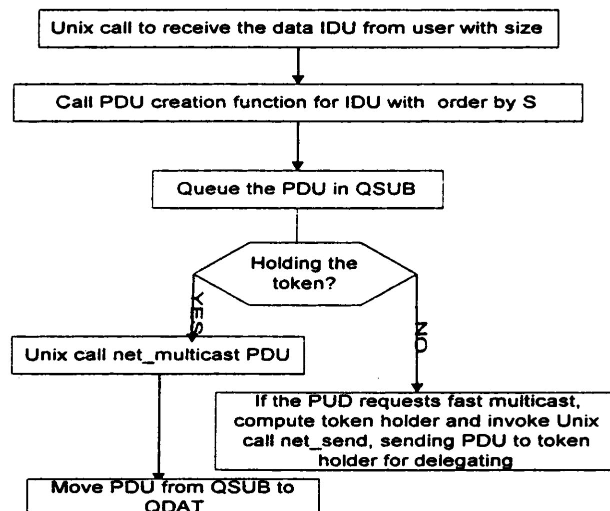

Figure 5. Steps taken for RMP to multicast an ordered message

Atomic protocol

RMP provides a consistent order and atomic deliveries: *consistent order delivery* is defined as a message m delivered by a process P on R in a consistent order if it keeps the total ordering of messages and the *version* of the ring is also kept. For instance, if a member receives two messages m and m' with $m.s<m'.s$ but `m'.version<m.version`, for this case to occur, even if the total order is preserved between m and m', the consistent order is still violated. *Atomic delivery* means that if a correct process P delivers m in the consistent order, it knows that all operational processes on R have received m and will deliver it.

The first approach achieving atomic delivery via delay does not need any network resource. When processes in the group received a message m from process P, they wait until the token goes through the entire ring R and back to P. At this moment, messages multicast by processes other than P have implicitly acknowledged m. Consequently, every process on R knows that all other processes have received m and will deliver it. This method is simple, but it also requires n–1 multicast delay time before m can be safely committed.

The second approach achieves fast atomic delivery via acknowledgment. To quickly achieve atomicity of a message, a scheme using a *safety parameter* has been proposed. Suppose P_i has multicast m. As soon as m is received at all the receivers, they sends their `ack`s to $P_{i\oplus 1}$ about the reception of m. Note that the messages multicast before m are also acknowledged. Upon reception of all `ack` messages, $P_{i\oplus 1}$ records `ack.s` in its `ack` array K, piggyingback $m'.k$, the safety parameter as the minimum value of $K[i]$, $i=\{0,\dots, n-1\}$. Atomicity is achieved by multicasting the next total order message m' that notifies all members that m is stable. It can be seen that only n–2 point-to-point `ack` messages are required to safely commit h messages, while h–1 messages are previously unstable messages sent before m. One of the important properties of the safety parameter is that every member process, on receiving a multicast message, can capture at least the current global view of the group without additional communication overhead.

The token process does not block itself from waiting for the `ack` message, which may not come due to a member process stopping. To keep the liveness of the protocol, the token process still transfers an ordered message even though the atomicity of previous multicast messages cannot be met at the moment. Token transferring to the failed member will trigger a fault tolerant mechanism of RMP, and the atomicity of the messages will finally be achieved, exclusive of the failed member, as illustrated in the next section.

Fault tolerant protocol

The fault tolerant protocol is designed on top of the membership protocol (see the next section). As described previously, the membership change is accomplished by the coordinator with the operational processes in two-phase communication. Identification of a fault is considered from two aspects:

(a) *Fault occurrence*: in normal operation, a single member P_i may fail at any time, but RMP treats the logical failure time as it happens when P_i serves as the current token process and no multicast message is ever received by any of the receivers on R on expiry of a certain timer, as described in the timer setting section.

(b) *Fault detection*: when there is a timeout, the rest of the members are able to detect a failure of P, even if P is a logical failure. During the ring reformation, the processes will see that P is still alive, and it can be included in the new ring.

Some optimized algorithms are presented here to deal with a single site crash without the election of a coordinator, so as to gain the performance optimization of the reformation. It is a common case that a single member may fail in a distributed system. In the presence of such a single failure mode, most existing protocols deal with the failure by electing a coordinator to form a new ring and eliminate the failed member. Normally, the reformation overhead is high. To authorize a reformation coordinator, several tests such as majority, sequence and resilience must be undertaken.[9] It may be that if a single process fails, repetitively, several processes compete to be the coordinator, trying to form a new membership list, and give up the reformation attempt later on because none of them could have recruited the majority of members. A substantial delay has been introduced.

The philosophy of the algorithm is to have the prior token process act as the ring coordinator and lead the surviving members to form a new ring when the current token process fails. In this way, the efforts to elect and authorize a coordinator are saved. More precisely, assume that the failure of current token process P_i is detected by some other member processes; those members will report the fault to P_{i-1}, if i=0 then to P_{n-1}, piggybacking their state such as the sequence number, etc. If P_{i-1} does agree with the fault reports, it serves as a coordinator to construct a new ring. There are several reasons for us to choose P_{i-1} as the coordinator: first, P_{i-1} has the highest probability of being alive since it just transmitted a message; second, if P_i fails before its transmission, the current message P_{i-1} is up-to-date, i.e. with the highest sequence number and safety parameter; and third, P_{i-1} can be reached by the other members with a higher probability because the other members just heard from it. Under the assumption of a single failure of P_i, P_{i-1} is the optimal choice to be the coordinator among the correct members without an election effort. The algorithm works by invoking the membership protocol, as shown in the next subsection.

Recall that the majority number is considered in terms of group size (N) instead of the ring size n and $n \leq N$. Suppose that the network does not partition and the failed member number exceeds $N/2$. When the multiple processes fail, in particular simultaneous failures of the current token process and the pre-authorized coordinator process, the single failure algorithm is no longer applicable. The surviving members have to enter into a complete reformation mode to elect a coordinator for the ring reconfiguration, which is accomplished by invoking the membership protocol.

When a network partitions, a group may be segmented into several small subgroups. The following cases must be considered:

1. Case 1. There is a subgroup with the majority of members of the original group. This subgroup will form a new ring. Any entering process can join the subgroup as long as the subgroup is reachable, that is, the communication paths exist between the join process and the operational subgroup. To cope with the reformation, a coordinator in the subgroup must be chosen in the steps of:
   (a) If the current token process does not crash and can be reached by the majority of members, the token process is the reformation coordinator;
   (b) In R the immediate prior process of P_i is defined as P_{i-1} (if i=0, then

P_{n-1}). If the token process crashes or is no longer reachable by the other operational members in R, the immediate prior process is the coordinator. This method is similar to the optimal algorithm for handling a single process failure, which saves the effort of selecting a coordinator.

(c) Recall that the majority membership is a necessary condition for group reformation. When (a) and (b) do not apply, the surviving members have to elect a coordinator to reconfigure a new ring by criteria R1 through R3. Those three protocols are also built on top of the membership protocol.

2. Case 2. There is no subgroup with a majority of members; all the processes in the subgroups are blocked. Since each subgroup will finally select a coordinator, the coordinators in the subgroups periodically broadcast an invitation message. Once the communication paths are reestablished, the subgroups can merge into a big ring containing the majority of members of the group. The protocol can resume its normal operation. The merging is similar to the algorithm of two candidates inviting each other in the membership protocol.

Membership protocol

The dynamic membership protocol allows a process to join or leave R dynamically, even if the process is not known in advance. Actually, RMP uses an Ethernet broadcast to implement multicast. On initialization, a new process executing RMP at any time is facilitated with an RMP port for messages. With the port, a process P outside R can detect the existence of R by listening to a message from R actively via this port. Once P receives a message m, it checks `m.version`; if it is greater than its own version, P picks `m.coordinator` as the ring coordinator and sends a new member request to the coordinator. Note that `m.coordinator` is the coordinator site id. The ring coordinator responds to the 'join' request and sends the current ring version and sequence number back. There are two ways for the coordinator to handle the new member join request: (1) it starts a reformation phase immediately; and (2) it waits until it is holding the token and then starts the reformation. A problem arises when the first method is deployed: if the coordinator immediately broadcasts an invitation message and enters into a reformation phase while some other sites are multicasting their normal ordered messages at the same time, the invitation message will interfere with members in normal operation. Some members may receive normal messages before the invitation and some after. Consequently, there is an inconsistent order of seeing the membership change interleaved with the normal multicast messages. To avoid such an interference, the second method is employed.

Holding the token and handling member-join: the coordinator keeps in mind the 'join' request until holding the token. At this moment, there is no other process transmitting normal messages and all members are expecting an ordered message from the coordinator, i.e. the token process. Subsequently, it starts a reformation procedure to include the applicants. This method will have the applicants wait for up to $n-1$ multicast transmission time but achieves non-interference. It is possible that a new starting process is impatient to wait for the reformation response of the coordinator and transmits its own ring invitation message. On receiving such an invitation, the existing ring coordinator and processes will respond with a negative

message that forces the process to give up the invitation. There are two cases must be considered: (1) the 'join' request is lost, and (2) the failure of the coordinator. In the former case, the new process normally sets a special timer to monitor the response of the coordinator. On timer expiry, the new process can retransmit its requests. In the second case of several time-outs without reception of the response from the coordinator, the new process will broadcast an invitation message. As mentioned before, this invitation forces the processes in the existing ring to respond. The new process is eventually able to join the group, along with the group reformation, as shown in the 2-phase algorithm below.

Member leave: in the case where a member voluntarily leaves the group, it simply multicasts a quit message and then leaves the group without having the permission of any other member. On reception of the quit message, the coordinator leads the other members to form a new ring. If the coordinator misses the 'leave' message, when the token moves to the position of the left process, the rest of the operational members will detect its leave because they are expecting an ordered message from the process. In this circumstance, the process is treated as a stopped process and RMP invokes the fault-tolerant protocol. Therefore, it is assumed that the 'leave' message has been received by the coordinator. A membership change in response to the 'leave' process is handled with a 2-phase algorithm:

Phase-1. The coordinator sends a 'reform' invitation message to all members and collects their `ack`s and message orders, i.e. value of `ack.s`, for an agreement about the reformation. In fact, nearly all the operational members send their `ack` to the coordinator simultaneously. If the coordinator sees any inconsistency between `ack.s` and its own *S*, it will do the retransmission of messages. Therefore, lost messages can be retransmitted. If the coordinator receives the `acks` from all the operational members or the majority `acks`, it goes into Phase-2; otherwise, RMP loops in this phase until the majority condition is met.

Phase-2. The coordinator constructs a new logical token ring R' in terms of the members acknowledged. It forms a message containing the new ring, multicasting the message, and piggybacking an up-to-date safety parameter to inform all the members to replace R and increment their ring version. Note that the messages sent in R should be committed despite the reformation. On receipt of R', all members on R' commit the messages in *QDAT* in accordance with the safety parameter. The coordinator multicasts a resume message to the members in R'. As soon as the resume message is received, all members deliver R' to the application processes, informing the users of the change of membership, and resume normal operation.

Note that the consensus of membership change is achieved approximately among the operational processes. A coordinator without recruiting the majority of members must block until some new processes join it, or it joins another ring. The blocked processes remain in the old ring in case several partitions occur. RMP allows users to decide if a minority ring can operate continually. In practice, as long as the majority of members are alive, the coordinator will finally form a ring with the majority of members, and RMP can continue its normal operation.

Dealing with failure during membership change: any member may fail during the reformation phase. Suppose a member, other than the coordinator, fails before its

sending *ack*. The coordinator can detect this failure by time-out, i.e. the coordinator is expecting an *ack* message from the process. On timer expiry, the coordinator eliminates the failed process out of the new ring. If the process fails after it sent out the *ack*, in this case, the coordinator cannot detect this failure. The failed process will be included in the new ring, but this causes no problems because normal execution of RMP in the new ring will eventually circulate the virtual token to the failed process. The failure can thus be detected by the fault-tolerant protocol described before. If the coordinator fails or is not reachable by the majority of members, a new coordinator must be elected according to the rules in the *ring creation* section. The rules are applied if a process receives more than one invitation, or if a candidate invites another candidate. As long as the coordinator is elected, the two-phase reformation algorithm can be performed again.

STATE MACHINE AND INTERFACE

Integrated RMP is implemented with a Finite State Automata (FSA) approach. A FSA transfers its state from one to another through events/actions interaction. Four FSAs constitute an RMP for the message multicast (**Send FSA**), reception (**Receive FSA**), group reformation and fault-tolerance (**Coordinator FSA** and **Reformation FSA**), respectively. RMP executes in either normal or reformation phase. On initialization, the processes enter the reformation phase to form a logical ring, and then switch to the normal operation phase. Once the group needs reconfiguring, RMP transfers from normal phase to reformation phase as illustrated in Figure 6.

An implemented FSA is designated as 2-d tables (**FSA_TAB**) indexed by events/states for the entries of action invocations. At any time, for a FSA in a

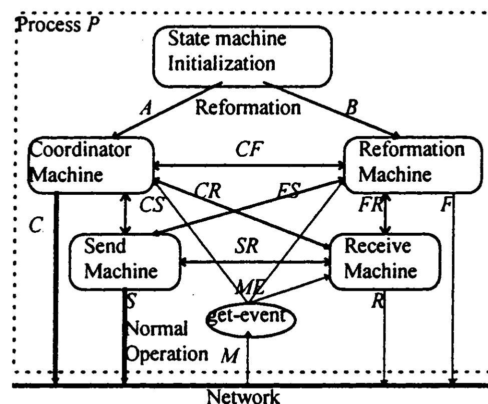

*Figure 6. FSA interaction: (*A*) On initialization,* P *acts as the coordinator if it first executes RMP. (*B*)* P *is not the coordinator and is prepared for joining in a new ring. (*C*)* P *multicasts an invitation or a new ring to the network. (*CF*)* P *transfers from a coordinator to the normal process, or* vice versa *(*CR, CS *and* FR, FS*) information exchange for reformation start or end between normal operation and reformation phase. (*F*)* P *is not the coordinator, it sends an* **ack** *message to the network. (*M*)* P *uses the interface procedure* **get_event** *to fetch packets from the network and interpret them into control information, and external events (*ME*) to trigger the machines, respectively. (*R*)* P *retransmits messages. (*S*)* P *holds the token and multicast (data) messages. (*SR*) The sending machine exchanges internal events with the receiving machine to notify each other that a message has been multicast or an expected message has been received*

specific state, its actions can be triggered by events. Each element in the table is an action entry. An event is the first index for the table and a state is taken as the offset of the index. Given an event and state, RMP is able to find the entry of the corresponding action and execute it. Program *rmp_main* calls the interface module *get_event( )* to fetch events as entries for invoking corresponding actions.

```
rmp_main( ):
 create and initialize FSA_TAB to include the pointer for FSAs;
 starting reformation by primitive = e_init and call action for
state machines initialization
 Create the ring R;
 while (TRUE)
 { clean packet buffer &msg;
  fetch event by primitive= get_event(&msg);
  if primitive=valid event then
   trigger the corresponding actions in FSA_TAB;}
}
```

Events

The events are categorized as internal, timer and external. The internal events are generated by FSA actions, denoting the control flow between the FSAs in the RMP. A timeout produces timer events. On reception of a message, its code in the **PDU_HEADER** is returned as the external events. The events are cited in Figures 7–12.

External events are generated from messages as control and data information:

1. **MEMBER_JOIN** (new member join request);
2. **FAST_ACK** (*ack* for achieving atomicity);
3. **REF_INV_BC** (reformation invitation);
4. **RESUME** (resumption message of coordinator);
5. **RESUME_RQU** (request the resumption message);
6. **RSP_POS** (Positive *ack*);
7. **RSP_NEG** (Negative *ack*);
8. **NEWTR_ACK** (*ack* for the reception of a new ring);
9. **ABORT_BC_RQU** (reformation abortion);
10. **COORD_RQU** (request the receiver to be a coordinator);
11. **TS_ALIVE_RQU** (ping ‘Are you alive?’);
12. **TS_ALIVE_RSP** (‘Yes, I am alive’);
13. **NEWTR_BC** (message for new token ring).

And data messages:

14. **RB_DATA_RQU** (Application data for multicast request);
15. **DATA_BC_RQU** (multicast request from another process);
16. **TOKEN_(data)BC** (total ordering multicast message, may be **NULL**);
17. **RESEND_DATA** (retransmission of lost messages).

Internal events, denoted by lower case, are coded corresponding to external events:

18. *e_ack* (expected multicast message has been received);
19. *e_coordinator* (the process assigns itself as a ring coordinator);

20. `e_init` (the process starts a reformation);
21. `e_abort_ref` (the process aborts its own reformation);
22. `e_refstart` (the process is informed the start of reformation);
23. `e_refend` (the process is informed the end of reformation);
24. `e_sendable` (the process is informed to send its multicast message).

Timer setting and timeout events

In asynchronous systems, using a timeout to detect failure is inaccurate. As a result, δ (Δ), described in the *assumptions* section, is used as an approximation for fault suspicion. As stated above, the basic message transmission timers are characterized as point-to-point message delay δ and multicast delay Δ. Specifically, several timeouts for the communication party are designated. Normally, a token process needs some time to maintain its state, such as timestamping, message storing, packaging or parameter calculations. Let the token processing time be T. Any timer expiring generates a corresponding timer event to trigger a specific action:

(a) **T_SEND** (event: **T_SEND_EXP**) is set by a token process to expect a multicast request from its applications or from other processes when $QSUB_i=\emptyset$. It should wait at least δ units so that a point-to-point request from other processes can reach it. Let **T_SEND**= $\Gamma+\delta$ **denote the timer; in addition to** δ**,** Γ **represents the extra waiting time of the token process for a request to arrive in the case of any delay. When T_SEND** expires, the token process multicasts an ordered **NULL** (live) message to make the protocol progress, as shown in the *ordered protocol* section.

(b) **T_RT** (event: **T_RT_EXP**) is set for receivers and **T_RT**=$\delta+\Delta+$T. All non-token processes on *R* use this timer setting for an expected ordered multicast message to arrive. They wait for a request to arrive at the token holder (δ), to be processed (T) and multicast back (δ). If they hear nothing from the token process on timer expiry, this indicates that either the message is lost or the token process stops. The potential receiver can choose to send a request to the token process. If a sender, during *C* successive retransmission attempts, receives no message from the current token process, it assumes that the token process may have failed. *C* represents the 'patience' of the processes. To make a request robust, one could choose a large value of *C* with a longer message delay.*

(a) **T_REFORM** (timeout: **T_REFORM_EXP**) is set to wait for an invitation message from some pre-existing coordinator process. On timer expiry, the process votes itself as the coordinator and multicasts an invitation message.

(b) **T_REF_RT** (timeout: **T_REF_RT_EXP**) is set by a non-coordinator process to expect a new logical token ring from the coordinator.

(c) **T_TS** (timeout: **T_TS_EXP**) is set by coordinator, waiting for acknowledgments from the rest processes.

---

* RMP sets *C*=2, **T_SEND** = 0.5 second and **T_RT** = 1 second.

Event generation interface

Event scheduling. Currently, RMP treats the events in mutual exclusion, i.e. when an event is processed by a process and another event for that process is blocked. The interface module checks the events in the order of internal, timer and external. Therefore, the internal events have a higher priority over others because an internal event is generated by an action as a control flow to trigger another action. For example, when the expected message is received by the *Receive* machine of process *P*, if *P* is holding the token, the module pushes an internal event to trigger its *Send* machine to multicast message immediately. This message multicast should not be delayed because other processes in the group are expecting the multicast.

Checking timer is taken before message reception, because a message size can be very large and message processing can be time consuming. For example, suppose *P* is too busy to respond to a request for message retransmission. Its timer for sending an order multicast message is expired; as a consequence, a longer delay is introduced for the total order multicast. More seriously, it may be that another process is expecting the message from *P* timeout and reports a fault of *P*.

Interface module: the internal events are pushed in a stack and popped upon invocation of the interface module `get_event( )`. It checks the internal event stack, the timer, and any message from local processes or the network, and returns an event to drive the corresponding actions. The module is the core of RMP, and is implemented as follows:

```
get_event(msg): union msg_pdu *msg;      /* message PDU buffer */
{ extern int sock_local,                 /* Local socket descriptor of users */
   int sock_network;                     /* UDP socket descriptor of network */
   NO_EVENT = TRUE;
   while (NO_EVENT) do                        /* get event until success */
   { if ((event = pop(stack)) not= -1)        /* check stack for internal events */
     NO_EVENT = FALSE;                        /* get an internal event */
     else                                     /* no internal event */
     { if (event = timer_check( ))            /* check if any timer expires */
       NOEVENT = FALSE;                       /* one of the timer expires */
     else                                     /*  no  time-out  and  no  internal
                                              event */
     {cc=select(sock_network, 0,0,0,timeout); /*UNIX call, check sockets of net-
                                              work */
      if (cc>0)                               /* there is a message arrives from
                                              the socket */
       {cc = net_receive(s_udp, msg);         /* peer the incoming message */
        event = pdu_match((PDU_HEADER *)msg);
                                              /* encode event for the message */
        NOEVENT = FALSE; }
      else                                    /* No message from network */
       {cc=select(sock_local, 0,0,0,timeout); /* check local user sockets */
        if (cc>0)                             /* there is a request arrives from
                                              local users */
        local_look(s_local, msg)              /* get the local request */
```

```
          event = DATA_RQU;                    /*get the event */
          NOEVENT = FALSE;
         }                                     /* end check local socket */
        }                                      /* end check network socket */
       }                                       /* end check timer */
     }                                         /* end check internal event */
return(event);
}                                              /* end get_event */
```

Note that the functions *net_receive( )* and *local_look( )* in this interface module make use of Unix system calls of *recvfrom( )* and *read( )* to receive messages from the network and local application, respectively. Function *pdu_match( )* casts the network message *msg* as the message header, checks its type and returns the corresponding external event. The function is implemented as follows:

```
pdu_match(msg)
PDU_HEADER *msg;                          /* incoming PDU header */
{ if (msg->code > 0)) AND (msg->code<=max_number_of_events)
  return (msg->code);                     /* return valid PUD code as event */
  else
   return(INVALID_PDU);                   /* an invalid PDU is returned */
}
```

When **INVALID_PDU** is returned, RMP gives a warning message to users that an invalid event is detected. This invalid event indicates that there are some non RMP messages intruding into the protocol. Normally, RMP ignores these messages.

Timer management: RMP has been provided with a very simple timer management module. Intuitively, the solution proposed does not seem very efficient. Most implementations define a timer agency, which orders the timers by their deadline in a queue; only the first timer in the queue needs to be evaluated at each clock tick. But for the case of RMP, as stated, five timers have been designated in RMP. **T_SEND** notifies the token process to expect a multicast request from its applications or from other processes if $QSUB_i = \emptyset$. **T_RT** is set for the receivers to expect an ordered multicast message to arrive; **T_REFORM** is set by a non-coordinator to wait for an invitation message from some pre-existing coordinator process. **T_REF_RT** is set by a non-coordinator process to expect a new logical token ring from the coordinator, and **T_TS** is set by the coordinator to wait for acknowledgments from the rest of the processes. It can be seen that at any time, for process *P*, it can only be in one of the machine states, but not in two. The state machines are disjoint, as are the timers. As described in the next section, when *P* transfers from one machine to another, the timer set in that machine is either expired or set off. Therefore, only one timer is on at any given time. Considering this fact, we thus make use of the simple approach for timer management in the interface module of RMP. For implementations of more complex protocols, such as a multiple group in wide or metropolitan area networks, this approach is not efficient. A more sophisticated method must be applied.

IMPLEMENTATION

At first, a state machine in RMP contains more than three states for handling the different events, but we have found that more states will cause complexity and may introduce a longer delay because more internal events may be generated. After careful design, RMP is reduced to three states for each machine.

Message send/receive

Send FSA

For the efficient design of the FSAs, `Send FSA` is not allowed to receive messages directly from the network. All normal message receptions are handled by *Receive FSA*. Three states comprise **Send FSA**: the initialize state *init*; *normal* for message multicast; and *waitack* for waiting for the expected message. The FSA is in the *init* state for reformation, and once the token ring is ready, *Send FSA* sees an internal event (*e_refend_s*) and opens a communication (TCP) socket to local applications. The TCP sockets enable the process to accept application messages and send them as ordered multicasts over network. If there is a request from an application in the state, it has to queue the requests. In the normal state, **Send FSA** is allowed to multicast a totally-ordered message on holding the token. Otherwise, a multicast request can be sent to the current token holder. After multicasting a message, the token has been transferred implicitly to the next member on *R*, and **Send FSA** enters the state *waitack*, waiting for the next multicast message from the token holder. When **Receive FSA** has received the corresponding message, it generates an internal event *e_ack* that triggers the state of **Send FSA** back to normal. Timer **T_SEND** is set to monitor the next totally-ordered message. If it expires and **Send FSA** does not hear any expected message, it sends the packet **TS_ALIVE_RQU** to ping the current token holder. If there is still no answer, a failure of the token holder is detected. **Send FSA** enters *init* for the reformation of the ring. The automata algorithm is shown in Figure 7. The initialization state responds with two internal events: reformation end (*e_refend_s*) and start (*e_refstart_s*) pushed into the event stack by the coordinator and **Receive FSA**, respectively.

Receive FSA is in charge of receiving ordered multicast messages and their delivery. Upon reception of the current token multicast message, the FSA checks the message validity, i.e. the version, ordering and membership of its sender, etc. It appends the message at the end of *QDAT* and updates the total sequence number *S* and ring corresponding entry. An out of order message is buffered separately in *QSUS*. Then the FSA enters into the *Resend* state, and requires lost message retransmission by sending a request message to the process that just transmitted the multicast message.

A reformation invitation is also handled by *Receive FSA*. Therefore, it has to decide the validity of a reformation invitation according to rules (R1–R3), and sends a positive (negative) *ack* to the coordinator candidate. If the process decides to elect itself as the coordinator, in this case, **Receive FSA** generates an internal event *e_coordinator* to notify *Send FSA* to switch to *Coordinator FSA*, or to *Reformation FSA* instead, as shown below. The states of **Receive FSA** includes initialization *init*, *normal* for message reception and delivery, and *Resend* for message retransmission.

```
State = Init (Initialization):
        if event=e_coordinator then
                inform Receive FSA about ring reformation;
                switch into Coordinator FSA (see Figure 11);
        if event= e_refend_s then
                State←normal (enter Normal state);

State = normal:
        if i=Si⊕1 (on holding the token) then
                Si← Si+1;
                Calculate safe parameter m.k← min(Ki), i∈ (0, ..., n-1);
        if QSUBi≠∅ then
                multicast m = head(QSUBi) and m.s = Si to R;
                QSUBi← QSUBi–head(QSUBi);
        if QSUBi = ∅ then
                wait for a request by timer T_SEND;
                multicast m.data=NULL and m.s = Si to R if no request arrives;
        QDATi←QDATi+m;
        if i≠Si⊕1 (not hold the token) then
                if QSUBi≠ ∅ contains urgent data message then
                        send m = head(QSUBi) with m.s=Si to PSi⊕1
        State ← waitack (wait for expected message);

State = waitack:
        if event = e_ack (received the expected multicast message) then
                State←normal; (back to the normal state)
        if event=e_coordinator or event=e_refstart then
                switch into Reformation/Coordinator FSAs (see Figures 10, 11);
```

Figure 7. **Send FSA** *for* P_i

State machines for group reformation

Reformation FSA. Internal event `e_init` generated by the fault-detection mechanism triggers the reformation automata. **Reformation FSA** checks if any reformation coordinator exists. If there is no such coordinator, it waits for a timer **T_REFORM** to see if it can receive any invitation PDU. On timer expiry, it votes itself as the coordinator and enters into the coordinator FSA by pushing an internal event `e_coordinator`. If there exists a coordinator, it responds with a positive `ack`, applying to join the ring. On reception of a new token ring, it opens a ring buffer, storing the new ring and resuming normal operation, as illustrated in the reformation algorithm (Figure 10).

Coordinator FSA. Initially, a coordinator is needed for a group of processes cooperating to form a logical ring. A competition occurs when there is no predefined group coordinator. In particular, every process stays in its initial state, voting itself

Upon reception of message msg:
State = Init (Initialization):
  if *event*= *REF_INV_BC* (reform invitation) **then**
    switch into Reformation FSA (see Figure 10);
  if *event*= *e_refend* (reformation end) **then**
    State ← normal (normal state);

State = normal:
  if No message arrives **then** set timer T_RT;
  if *msg.*sender=$P_{Si\oplus 1}$ and *msg.s*=S_i+1 (correct order msg) **then**
    S_i ← S_i+1; $QDAT_i$ ← $QDAT_i$+msg;
    push internal event e_ack to notify send FSA;
    commit (atomic) *msgs* ∈ $QDAT_i$ to application in terms of *msg.k*;
  else if *msg.s* ≠ S_i+1 (out of order msg) **then**
    $QSUS_i$ ← $QSUS_i$+msg;
    send a request to *msg.*sendre for message retransmission;
    State ← Resend; (wait for the retransmission)
  if S_i > *msg.s* (*msg.*sender lost messages) **then**
    send msgs ∈ $QDAT_i$ to *msg.*sender;
  if *event*=*T_RT_EXP* (timeout) **then**
    execute Fault-detection on $P_{S\oplus 1}$;
    push(e_coordinator/e_init) if the fault is confirmed;

State = Resend:
  if received correct retransmission **then**
    link $QDAT_i$ and $QSUS_i$ by ($QDAT_i$+$QSUS_i$);
    push e_ack to notify Send FSA the reception of expected message;
    State ← normal;

Figure 8. **Receive FSA** *for* P_i

as the coordinator and intending to form a logical ring. Selection rules (R1–R3) are applied to elect the coordinator quickly to reduce the reformation time, and to avoid any race. After the election, the coordinator constructs a new token ring and multicasts the ring across the group. If the new ring is acknowledged by the members, a resume message is sent and the FSA switches to a normal phase. The coordinator algorithm is shown in Figure 11. Figure 12 shows the events/Action combinations for the Reformation and Cordinator FSAs.

PERFORMANCE

RMP is implemented on a cluster consisting of Sun 4C/Sun 4M workstations (MC68020–30s, 16.67–20 MHz) connected to a 10 Mbit/s Ethernet by an AMD (Advanced Micro Devices) Lance chip interface. The sites used in the experiments were able to buffer 32 Ethernet packets before the Lance overflowed and dropped

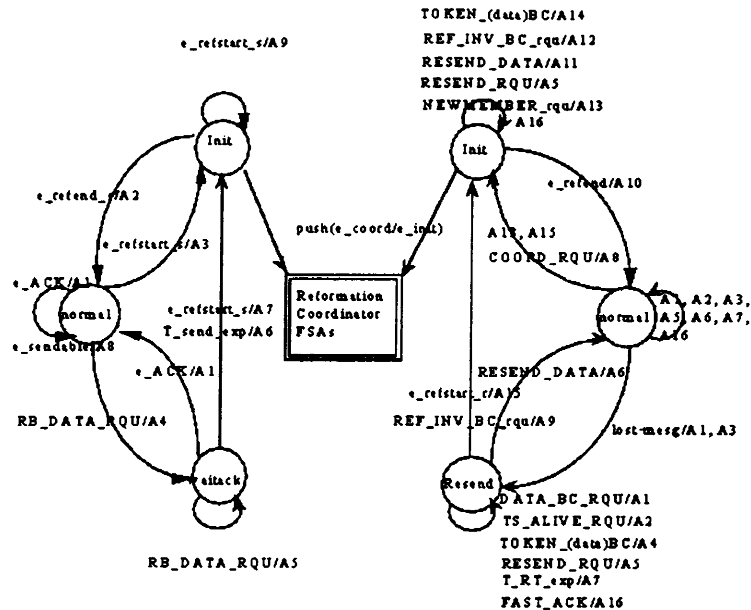

Figure 9. FSA state transition graph in which `Send FSA` *is on the left and* `Receive FSA` *is on the right.* `Ax` *(where x = 1,2,3...) is the same as* `event/Ax`*, e.g.* `A1` *is the same as* `DATA_BC_RQU/A1`*. Detailed descriptions of the actions are omitted here, and their roles have been described in Figures 7 and 8*

packets. Most measurements were taken with message sizes from a null packet to 4 Kbytes, exclusive of the RMP message header of 64 bytes.

Overhead

An initial set of experiments measures the overhead in performing several performance-critical operations. The overhead of the operation is measured in a light-weight network by transmitting `NULL` size messages. The results are presented in Table I. Three factors are of primary importance: first, the user call of *rmp_recv/mcast*, the overhead to entrust the data to RMP, which causes a context switch (about 50 μs) and managing its own message buffer (about 130 μs); second, the overhead caused by the RMP interface due to the checking event stack, timer and port signals for arriving messages; and third, the delay of transmitting a packet over Ethernet depends upon the packet size and usage of network.

Delay analysis

This section uses approximation analysis to compute the communication delay. Four parameters are particularly important for the delay analysis:

1. T_{copy}: the typical time required to perform a generic copy or calculation;
2. T_{setup}: the typical time required to set up a communication by TCP/IP;
3. $T_{pp\text{-}comm}$: the typical time taken to communicate a single atom data (a byte) between any two connected processes;

```
State = Init: (reformation stage)
        if there is no Coordinator then
                set timer T_Reform for coordinator invitation;
        if event=REF_INV_RQU (received an invitation msg) then
                Respond msg.sender in lines of (R1-R3);

        if respond positive ack then
                State←W_Tr (wait for new logical ring);
                Set timer T_Ref_RT waiting for the New Token Ring;
        if respond negative ack then redo the reformation;
        if event=T_Ref_RT_exp (timeout) then
                Vote self as the coordinator by push(e_coordinator);
                switch into Coordinator FSA (Figure 11);
        if event=ABORT_BC_rqu (reformation abort) then
                switch into Normal operation by push(e_refend);
State = W_Tr:
        if event=T_Ref_RT_EXP (There is no New Ring received) then
                State←Init;
        if event = NEWTR_BC (received New Token Ring) then
                accept (set) the New Ring and acknowledge the coordinator;
                State← W_rsu (waiting for the final resume msg);
        else    State←Init; (restart the reformation)

State = W_rsu:
        if event=TOKEN_RESUME (received valid resume) then
                Install new ring and deliver the ring to applications;
                State←Init; switch into Normal operation FSAs (see Figures 7 and 8)
```

Figure 10. **Reformation FSA** *for process* P_i

4. $T_{nn\text{-}comm}$: the typical time taken to communicate a single atom data (a byte) between any two connectionless processes over a network.

The magnitude of T_{copy} depends upon both the nature of the operation and upon such things as the storage location of operands. On setting up a connected communication, communication hardware and software incur significant set-up times. RMP provides flexible communication—both connection and connectionless are used for message transmission. Based on our experiments on Sun workstations, we have shown that the communication parameters between applications and RMP processes are $T_{pp\text{-}comm} \cong 0.62\ \mu s$. The approximate communication setup times are measured as $T_{setup} \cong 130\ \mu s$ (setup connection between user and RMP processes via a TCP port), and $T_{copy} \cong 0.34\ \mu s$ for copying a 1 byte message. The time to transfer one byte over 10 Mbit/s Ethernet is taken as $T_{nn\text{-}comm} = 0.8\ \mu s$.

In the following, function $T(X)$ is used to denote the times used to process a message in layer X, and $T(X-Y)$ to denote the times used to pass a message through

```
State = Init:
    if event=coordinator (I am coordinator) then
        broadcast an invitation msg;
        set timer T_TS for collecting response;
        Create a New logical token Ring R and id(P)<-P0;
        R[0]<- P0; i<- 0; State<-W_rsp (wait response state);
    if event= ABORT_BC then
        switch into Normal operation FSAs; reformation end;

State = W_rsp:
    if event=T_TS_exp timeout then
        if received majority positive acks (i≥⌊N/2+1⌋) then
            multicast New Ring R;
            State<-W_ref; (wait for acknowledgment of reformation)
        else multicast an reformation invitation and set T_TS;
    if received positive ack then
        id(ack.sender)<- Pi; i++; R[i]<- Pi;
    if received a negative ack then
        vote ack.sender according to R1-R3;
        notify all members on R that new coordinator is ack.sender
        State<-Init; switch into reformation FSA ;
    if received all positive acks (i=N-1) then
        multicast new token ring R;
        State<-W_ref; (Waiting acknowledgment state)

State = W_ref:
    if received acks from the members on R then
        multicast resume message;
        switch into Normal operation FSA (Figures 7 and 8);
```

Figure 11. Coordinator FSA *for process* P_i

layer X to layer Y. For each user application, communicating a 1 Kb message (IDU) from the user to the RMP incurs the communication time

$$T(user\text{-}RMP) = T_{setup} + 1000 * T_{pp\text{-}comm} = 130 + 620 = 750\ \mu s \quad (1)$$

In general, to copy the 1 Kb message from the user to process P_i requires cost of

$$T(RMP\text{-}user) = T(RMP\text{-}interface) + T(copy) = 180 + 1000*0.34 = 520\ \mu s \quad (2)$$

If one considers the time to packet and store (copy) the IDU in a queue QSUB to become a PDU, this method suffers another copy time of

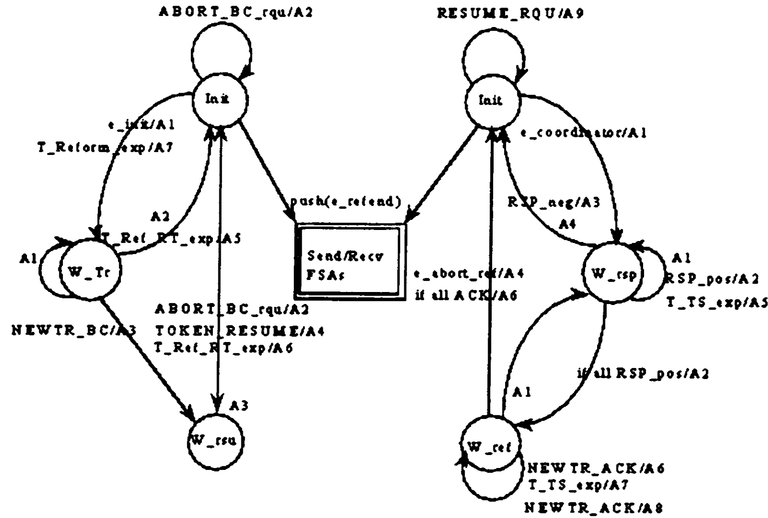

Figure 12. The Reformation (left) and Coordinator (right) FSAs show the events/Action combinations. Note that detailed descriptions of the actions are omitted

Table I. Cost of the various layers of system

Layer	Time (μsec)
user call rmp_r/m	130
RMP interface	180
RMP packeting	69
IP + UDP	78
RMP msg buffering	72
Ethernet	269
Total	791

$$T(\mathit{RMP\text{-}buffering}) = T(\mathit{RMP\ packeting}) + T(\mathit{copy}) = 69+1000*0.34 \cong 410\ \mu s \quad (3)$$

We consider the times taken for RMP to prepare the multicast of a 1 Kb message and copy it to the UDP for multicast. Using Ethernet to transmit the message over UDP/IP incurs an overhead of

$$T(\mathit{network}) = T(\mathit{IP+UDP}) + T(\mathit{Ethernet}) + 1000*T_{nn\text{-}comm} = 78+269+800 \cong 1.14\ ms \quad (4)$$

Thus,

$$T(\mathit{RMP\text{-}network}) = T(\mathit{RMP\ buffering}) + T(\mathit{network}) = 410+1.14 \cong 1.54\ ms \quad (5)$$

Therefore, the total time for the user-to-network delay would be

$$T(user\text{-}net) = T(user\text{-}RMP) + T(RMP\text{-}user) + T(RMP\text{-}network) = 750 + 520 + 1540 \cong 2.8\ ms \quad (6)$$

If 2.8 ms is used for the estimation measurement for a process to receive a 1 Kb message and deliver it to the user, the cost would be 5.6 ms for a single user-to-user multicast in the normal case for a single 1 Kb message. Likewise, the overhead for a control message can be calculated. To estimate the delay, the control message is considered as very light. The costs of message packeting, IP, UDP invocation on Ethernet can be summarized as 416 μs. To achieve an atomic message, by the atomic protocol, the total delay would be a user-to-user delay (5.6 ms) plus one delay of `ack` (control message delay 0.416 ms) plus another multicast message (2.8 ms), a total of about 8.8 ms would be required for a single 1 Kb atomic message in a fault-free network. Taking this analysis into consideration, assuming the case of continual multicast messages, the times for message multicast and message reception can be overlapped between any two processes. Thus, half the times are used for calculation of the throughput, i.e. 2.8 ms delay for total order messages and 4.4 ms for atomic messages. A protocol is able to achieve throughputs of 357 1 Kb messages/s for the total order delivery and 227 1 Kb messages/s for atomic message delivery.

Optimization of message processing

To reduce the overhead, we have chosen to re-implement the core of RMP. Since the overhead *T*(*user-RMP*) and *T*(*Ethernet*) are not changeable, the times of *T*(*RMP-user*) and *T*(*RMP-buffering*) are the factors of consideration for time reduction. Since *T*(*RMP-user*) contains two items, *T*(*RMP-interface*) and *T*(*copy*), our strategy is to minimize the message copy in RMP by sharing the buffer between the user-RMP and RMP-UDP interface. To achieve this goal, RMP uses a fixed buffer size (1.5 Kb for Ethernet, which may change for different network settings) for the message receptions either from a user or the network. RMP always has a buffer (1.5 Kb) ready for each user, and a pointer to the buffer is provided for an application in the RMP interface. Therefore, users can copy their data directly to the buffer of RMP. On reception of a message, RMP also directly saves the message in a buffer without an extra copy. Consequently, *T*(*copy*) can be skipped and *T*(*RMP-user*) is reduced as a constant of 180 μs.

On the other hand, RMP does not copy the PDU in QSUB for multicast. It only packets the message from IDU into PDU by adding the *header* before the IDU pointer, and it uses the PDU pointer for the communication. In formula (3), *T*(*RMP-buffering*) is again taken as a constant value of 69 μs, which is a measurement for a single message. In the case where RMP multicasts messages continually, a user's message sending can be performed in parallel with the RMP's message multicast. Therefore, the time of *T*(*user-RMP*)=750 μs can be overlooked. Formula (5) becomes

$$T(user\text{-}net) = T(RMP\text{-}user) + T(RMP\text{-}buffering) + T(network) \quad (6')$$
$$= 180 + 69 + 1140 \cong 1.4\ ms \text{ for a 1 Kbyte message}$$

which is about a 50% performance improvement over the previous analysis.

RMP uses the same method to deal with the message reception from a network. The delay of 1.4 ms is taken as an approximate measurement for a process to receive a 1 Kb message and deliver it to a user in a consistent order in the normal case. The optimized throughput is about 715 messages/s. To achieve an atomic message, by the atomic protocol, the total delay would be a user-to-RMP delay (1.4 ms) plus one delay of `ack` (control message delay 0.416 ms), plus another multicast message (1.4 ms), which is about 3.2 ms for a 1 Kb atomic message in the fault-free network. Therefore, RMP is able to achieve throughputs of 571 1 Kb messages/s for consistent (total) order delivery and about 250 1 Kb messages/s for atomic message delivery.

Experiment performance

The experiment is as depicted in Figure 13, in which two senders continually send **Null**, 1 Kbyte and 4 Kbyte packet multicast messages. In this figure, since user messages are generated continually, the times of *T*(*user-RMP*) are overlapped with RMP multicast communication. Considering this fact, we overlook the time *T*(*user-RMP*), therefore for a 0-byte message the delay is calculated as

$$T(\mathit{user\text{-}net}) = T(\mathit{RMP\text{-}user}) + T(\mathit{RMP\text{-}buffering}) + \mathrm{T(RMP\text{-}network)} = 0.67\ \mathrm{ms} \quad (7)$$

This result contributes the analysis of throughput of RMP 1497 0-byte messages/s. In the experiment, for a group of five processes, RMP can achieve a peak throughput about 1400 ordered messages, which matches the analysis result perfectly. It is higher than that of any previous protocols. For the recent published protocol Totem,

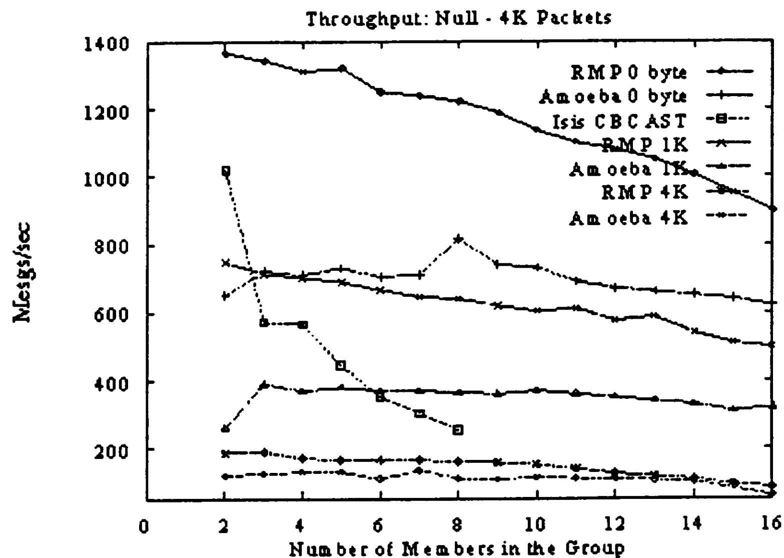

Figure 13. The number of ordered multicast messages per second of RMP. The performance of Amoeba is taken from Reference 21, Figure 4. The figure of CBCAST is taken from References 8, Table 12.2

its throughput is less than 1200 agreed (totally-ordered) messages with the same hardware and network settings.[6]

We have compared our experimental results with the Isis CBCAST[8] and Ameoba[21] protocols under the same hardware setting. Note that in Isis, CBCAST is faster than ABCAST. Figure 13 indicates the committed message number received by application programs with total order, showing the maximum throughput of RMP. The reasons that RMP is efficient are threefold. First, all members are able to locate the token position quickly; in this way, total ordering of multicast messages can be achieved efficiently among the group of processes. If no message is lost, all members in the group are able to locate the token holder, and monitor the expected message from the current token process unanimously. Second, if any message gets lost, a process receiving the unexpected message will help it to detect the number of missed messages. Third, RMP is implemented to minimize the message buffering and copying, which is an important improvement over message delay as it is caused mainly by message processing instead of transmission.

In the experiments, a short logical token cycle time has been set and two senders each send 2000 fast multicast messages continually. If the token process transmits a message from its own applications, the messages are multicast without any delay. In the case of multiple senders, one of the senders may have to retransmit its requests, but the request is still taken as an acknowledgment message. As the message load becomes higher, if every member transmits its own application message periodically, the number of extra control packets drops to zero. In this experiment, for a group of three processes, the measured delay for a 64-byte message is about 1.5 ms, only the message header of 64 bytes. For a group of 16 members, the delay is about 2.6 ms. When the message size increases to 1 Kbyt, the delays are 2.9 ms for a group of three members and 4.5 ms for a group of 16 (Figure 15). Those message delays are measured through user-to-user communication for individual messages.

Note that the performance shown in the figures is not the best performance of

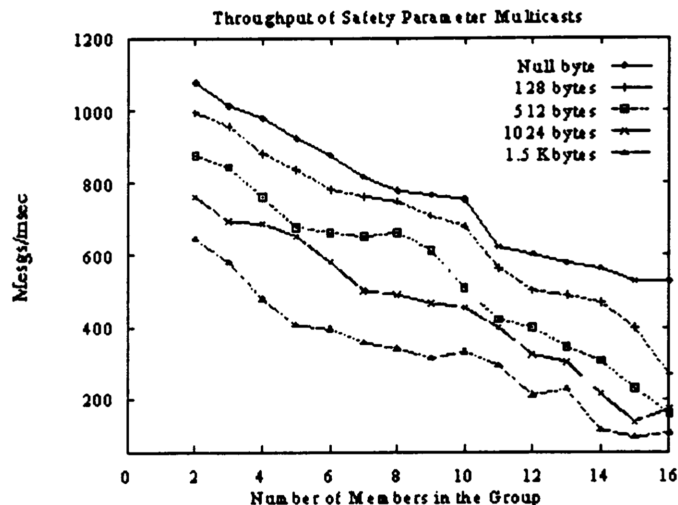

Figure 14. Throughput of atomic multicast messages with safety parameter

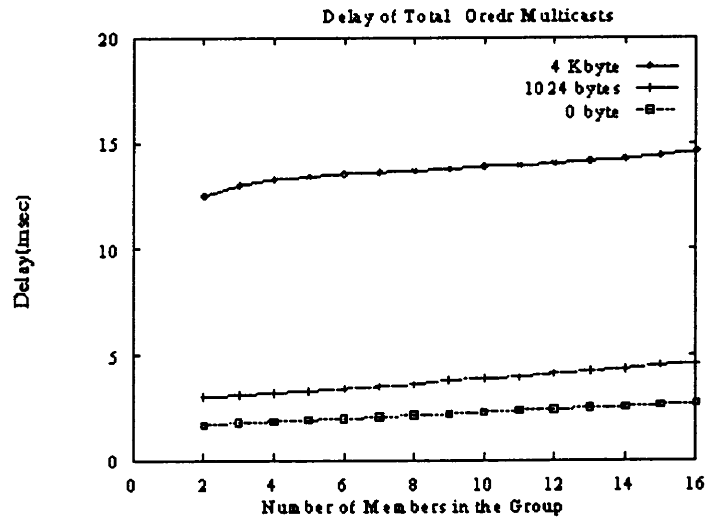

Figure 15. Delay of total ordered messages

RMP. Using two senders to multicast burst messages (e.g. each sends 2000 1 Kb messages continually) causes problems of many message collisions and retransmissions. When both senders do not hold the token, they may send their messages as requests to the token holder simultaneously, and one of the requests has to be retransmitted. For a large group, message retransmission causes more performance penalty. If each member sends messages, message multicast can be executed on holding the token. The performance can be further improved.

With the safety parameter and advanced acknowledgment approach illustrated in the *atomic protocol* section, the performance of RMP in terms of atomic delivery delay is excellent (Figure 16). This is because the acknowledgments of a multicast

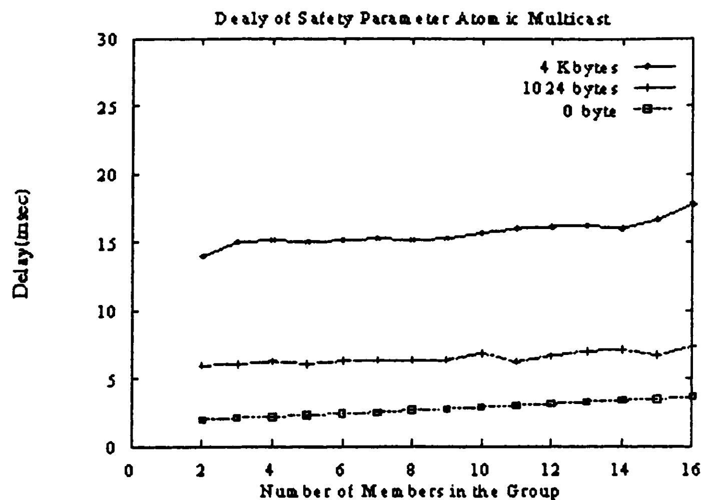

Figure 16. Delay of atomic messages by safety parameter

m are sent to the next token process immediately after *m* is received by the receivers. At most N–2 point-to-point `ack` messages are transmitted in the network. Assume that there is no message lost; the next multicast message piggybacking the safety parameter for *m* by the next token holder causes no additional communication cost. In the optimal case, only 1-phase delay is introduced (receivers sending 1 `ack` and accepting the next multicast message). The overhead for achieving message atomicity is reasonable, and RMP has achieved a good balance between the extra packets and delay.

Moreover, when a process in RMP has nothing to multicast, on holding the token, it has to transmit a **`NULL`** message. This consumes some network resources. Considering that the general 'load' is 1 multicast message/s when RMP has nothing to send, that is not the case for the protocols of Isis or Ameoba. In the case of sporadic messages, the delay of RMP is given in Figure 17, in which a longer time is required to deliver messages because of some retransmission and **`Null`** messages. The token is transferred more times due to **`Null`** messages. On the other hand, some application messages have to be forwarded to the token holder for a delegate multicast, and an extra point-to-point message delay is introduced.

RELATED WORK

A number of well-known systems have been published recently, addressing reliable message multicast, fault-tolerance and dynamic group for achieving a consistent global state of the cooperation processes. To make the comparison feasible, we restrict our comparisons mainly of the algorithms are designated for asynchronous computing environments.

Chang and Maxemchuck (CM)[9] describe a family of protocols that achieve total ordered broadcasts determined by a single process in a group of processes. This process is called a 'token site'. Initially, a sender broadcasts a message with a unique message id to all recipients, including the token site. If the token site receives

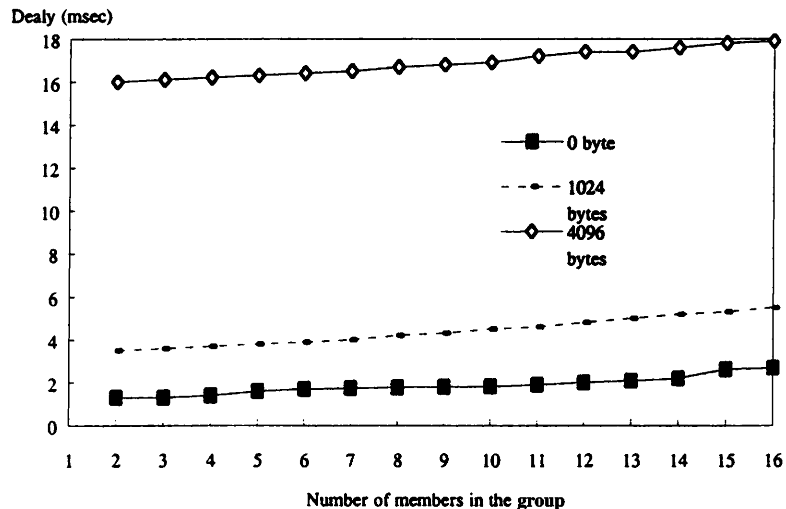

Figure 17. RMP delay when there is only sporadic message traffic from applications

the broadcast message, it broadcasts an acknowledgment message containing the message id and timestamp (total order). The CM protocol requires that the token be periodically transferred from site-to-site in a list (called the 'token list'). A message may be committed, and memory of it discarded only when the token has been passed twice around the site in the token list.

Birman *et al.*[18] developed Isis Toolkit in which a family of protocols is proposed and implemented under the names *CBCAST* and *ABCAST*.[7,8] The *causal broadcast* primitive *CBCAST* is used to enforce a delivery ordering when desired with minimal synchronization. *ABCAST* achieves total message ordering, and operates by assigning each broadcast a timestamp and delivering messages in the order of the timestamps. When a site receives a new message, it stores the message in a pending queue, marking it as *undeliverable.* It then sends a message to the initiator with a proposed *timestamp* for the broadcast message. This proposed timestamp is chosen to be larger than any other timestamp that this site has proposed or received in the past. The messages buffered in the queue are then reordered in terms of the large timestamp. Therefore, the sites are able to organize the message in the same relative order with other messages in the pending queue, and eventually make the message deliverable. The concept of *virtual synchrony* is proposed by Isis and used to guarantee the consistent group view change among the processes.

Melliar-Smith *et al.*[12] present two protocols which together implement an atomic broadcast facility. The *Trans* protocol is used for efficient reliable communication, and ensures that messages are eventually received at all destinations by piggybacking positive or negative acknowledgments. The *Total* protocol places a total order on the partial order messages achieved by *Trans*. However, since agreement about a total order can only be achieved after sufficient messages from different senders have been received, substantial delay in the case of spare message traffic are possible. To avoid these delays, additional *NULL* messages are sent by processors which did not transmit a broadcast for a long time.

Rajagopalan and McKinley[13] developed a protocol, TPM, based on the token ring approach. In their scenarios, a token is explicitly transferred from one process to the next on a token ring. Only the process holding the token can broadcast or retransmit messages. On receipt of the token, a processor completes the processing of messages in its input buffer, broadcasts messages, updates the token and transmits it to the next processor on the ring. Amir *et al.*[6,22] designed a token-ring-based protocol, Totem, which is similar to the TPM approach. The mean latency of Totem for safe delivery is faster than TMP, approximately two rounds of token rotation times. Both TPM and Totem achieve safe delivery but require, on average, two and a half token rotations for TPM.

Similar to the approach of the CM protocol, Kaashoek and Tanenbaum[10,21] simplified the token list approach by using a fixed sequencer for Amoeba systems. When the sequencer receives a point-to-point message, it allocates the next sequence number and broadcasts the message with the sequence number. The lost message can be detected by a gap in the sequence numbers.

Veríssimo *et al.*[14,23] described a protocol, xAMp, that achieves reliable delivery and consistent order of messages by a 2-phase commit protocol, which is controlled by the sender. In the first phase, a message is broadcast and all potential receivers send back an acknowledgment message which confirms receipt of the message. If all receivers have returned affirmative reply messages, the sender broadcasts an

`accept` message. Otherwise, a `reject` message is broadcast. The sender, after multicasting the message, sets a timer to monitor the expected `ack`s. In case any expected `ack` message is not received by the sender due to a receiver(s) stopping, the sender initiates the membership checking. xAMp provides reliable services so that the receivers also monitor the sender by timer setting to expect the `accept/reject` message. On timer expiry, a receiver pings the sender and a monitor action for the membership is invoked. Therefore, for xAMp, the delay between the sender and receivers is the timer duration of both sides.

Luan and Gligor[24] provided a 3-phase voting decentralized solution that relies on a majority consensus among designated processes at each site to commit on the ordering of broadcasts. Garcia-Molina *et al.*[25] have proposed an approach to solve the multiple overlapped group message-ordering problem. In their protocol, a tree is superimposed on the set of processes in the system. To transmit a broadcast, the message is forwarded to the least common ancestor of the destination processes, which in turn uses a reliable FIFO protocol to handle the message delivery.[25] There are also other solutions that we do not review here in detail. Reference 26 presents a solution to the multiple source ordering problem. Each multicast group has a group manager responsible for delivering the messages to the group members.

Modular versions of protocols developed recently include the Horus system,[18] the successor of Isis, which provides efficient support for the *virtually synchronous* execution model to achieve the agreement of membership change in the same relative order, and which has been adopted with some changes by such systems as Transis,[27] Trans/Total and Totem.[6,22] The model is based on group membership and communication primitives, and can support a variety of fault-tolerant tools. Horus provides a structured framework for protocol composition which incorporates ideas from such systems as the Unix 'streams' framework and the *x*-kernel[16] which is also used to build the *Psync*,[28] modular Consul[11] and micro-protocols.[17]

x-kernel is designed to facilitate the implementation of efficient communication protocols. It defines a simple interface between protocols that only includes operations for opening and closing connections, and for sending and receiving messages; additional operations, if needed, must be encapsulated as *control operations*. Such an interface has limited interactions, but is overly restrictive for fault-tolerant protocols, which interact much more closely. On the other hand, the *x*-kernel is designed primarily to support a hierarchical composition of protocols, i.e. a protocol stack where one high-level protocol depends upon one low-level protocol. In Consul and the succeeding approaches[11,17] several protocols at the same (logical) level cooperate to implement a set of services.

Horus provides a group that can be varied to match application requirements. It does this using a structured framework for protocol composition which incorporates ideas from *x*-kernel. Its system is thus like a 'box' of 'Lego' blocks with a set of entry points for down- and upcall procedures. In addition, Horus also introduces run-time configuration, a group communication interface and a full thread-safety.[18]

RMP is also modular in its sub-protocol design which is different from *x*-kernel and Horus in that it is a primitive-based, whilst the former is a functional-based, protocol built on top of Unix BSD. Therefore, *x*-kernel can be viewed as the 'primitive modular' and RMP is the functional modular, i.e. RMP uses state machines to assemble the functions of a sub-protocol. The significant difference of RMP from these modular systems is that the accomplishment of sub-protocols are executed in

a non-blocking way, i.e. achieving message atomicity can be done in parallel with message total ordering. Therefore, sub-protocols in RMP do not need to be synchronous between ordering and atomicity. Likewise, there is no synchronous requirement between membership change and fault-detection. On the other hand, the function invocations (sub-protocols as well) in RMP are not organized in a purely hierarchical structure. In Horus it is possible to dynamically create and add a process in a group, but in RMP a process is at a lower layer of an application and it serves as a reliable multicast server; thus, the process has to be known before it can be included in a group.

The inherent disadvantage of the token-based solutions is the delivery latency. In existing token based algorithms, the token is transferred explicitly in the membership list (logical ring). This scheme may bring a token loss problem due to missing the token passing message, or more seriously, the crash of a process which is going to transmit or accept the token. Consequently, some token information may not be recoverable, and the protocols have to enter the reformation phases. The reformation mode normally requires several rounds of message transmissions for electing a coordinator. On the other hand, token-based protocols occupy more memory space because all the messages are saved until the acknowledgments arrive or the token has rotated (a minimum of) two rounds along the ring in order to safely deliver the messages as shown in References 6 and 13. RMP only buffers n (the size of the ring) messages for possible retransmission, and one round of token passing is enough for attaining message safety. The Amoeba protocol does not require each site to buffer all the messages, only the sequencer (token) keeps the message history, but their protocol needs a synchronous phase for emptying the message buffer of the sequencer.

In normal operation, RMP has minimized the control message overhead: for each multicast message, only one point-to-point message is needed to transmit the request from a source process to the token holder with some `ack` information piggybacking. In the CM protocol all messages are broadcast, whereas RMP uses a point-to-point message whenever possible, reducing interrupts and context switches at each site. In the normal case, the CM protocol generates $2(n-1)$ interrupts for each multicast, ours only generates n, one point-to-point from sender to the current token and $n-1$ from the token to the receivers. In the case of a request collision or message retransmission, RMP still continues its data message multicast. The retransmission only introduces some point-to-point messages, however, they can also be taken as `ack` messages. This retransmission rate (r) varies with different applications, nevertheless, in the worst case, $n-1$ retransmissions of a request may happen. If the token holder transmits a message from its own applications, this message can be directly multicast without any extra point-to-point message. Therefore, RMP uses control messages r where $0 \leq r \leq n-1$.

RMP also incurs a delay to gain the safe delivery of a message. The design of a safety parameter and a committed array facilitates the safe delivery of messages. To meet the safety requirement of a multicast, the protocol waits for up to another $n-1$ multicasts to come and captures the global instant state of the group at the same time. By fast-type message commitment, RMP enables the members in a group to speed up the commitment of all previously received messages up to the current multicast. This mechanism greatly enhances the flexibility of the protocol. The latency for the fast type is two multicast transmissions: one from the token to the

receivers and another transmitted by the receivers. RMP does not have the token loss problem. Single site failure handling mechanisms can be invoked by a pre-recoganized coordinator (provided it is still alive). Thus, the expensive reformation can be optimized to a simple 2-phase communication between the coordinator and the surviving members. Although RMP also has a central process (token) per multicast, the protocol does not fully rely on a single token holder. All processes equally take the responsibility of being the token holder for message multicast. The approach of a token transfer implicitly per multicast message distributes the message transmission load over the members of a group, and effectively overcomes the bottleneck problem, even the group is large. If the application message load is light, RMP has to send *NULL* messages that use some network resources.

CONCLUSIONS

We have presented the implementation of RMP based on a state machine approach for general distributed asynchronous systems. RMP based on logical token ring has a very simple structure. The protocol requires a minimum number of control messages and transfers the token responsibility among the processes on a logical ring, placing total orders on each multicast message and ensuring the messages are received by all correct processes in the same order. RMP has provided efficient algorithms for reliable message delivery, ordered failure detection and recovery. In addition, it offers flexibility of dynamic group membership changes, requiring the minimum resources of the underlying communication networks. In respect of message atomicity, RMP has minimized the control messages and communication costs while incurring a short delay. It is superior to existing token-based protocols, and is competitive with the two-phase method in terms of safe delivery delay. RMP can be further optimized by avoiding any unnecessary **NULL** message multicast. Our experiences demonstrate that the implementation of a complex system such as RMP needs a modular and cooperative design. The significant contributions of RMP lie in its global optimization on the module design and implementation for total message ordering, atomicity, dynamic group configuration and fault-tolerance. Among the existing protocols in the same hardware and network settings, RMP has achieved excellent performance, as shown by the experimental results.

ACKNOWLEDGEMENTS

The author thanks E. Nett, J. Kaiser, M. Mock, M. Gergeleit, B. Weiler, R. Frings, A. Broehl, and the members of SET-RS, GMD, for their help during the initial implementation of the protocol. The author is grateful to the referees, who made valuable comments on earlier versions of this paper. This research is currently supported by City University Hong Kong grants Nr. 7000578, 9030464 and UGC HK grant Nr. 9040228.

REFERENCES

1. F. Cristian, 'Reaching agreement on processor-group membership in synchronous distributed systems', *Distributed Computing,* **4**, 175–187 (1991).
2. M. J. Fischer, N. A. Lynch and M. S. Paterson, 'Impossibility of distributed consensus with one faulty process', *J. ACM,* **32**(2), 374–382 (April 1985).
3. D. Skeen, 'Nonblocking commit protocols', *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1981, pp. 133–142.

4. W. Jia, J. Kaiser and E. Nett, 'An efficient and reliable group multicast protocol', *Proc. 2nd Symp. on Autonomous Decentralized Systems*, Phoenix, USA, April 25–27 1995, pp. 127–133.
5. W. Jia, J. Kaiser and E. Nett, 'RMP: fault-tolerant group communication', *IEEE Micro,* **16**(15), 59–67 (April 1996).
6. Y. Amir, P. M. Melliar-Smith, L. E. Moser, V. Agrawala and P. Ciarfella, 'The Totem single-ring ordering and membership protocol', *ACM Trans. on Computer Systems,* **13**(4), 311–342 (November 1995).
7. K. P. Birman and T. A. Joseph, 'Reliable communication in the presence of failures', *ACM Trans. on Computer Systems,* **5**(1), 47–76 (February 1987).
8. K. P. Birman, A. Schiper and P. Stephenson, 'Lightweight causal and atomic group multicast', *ACM Trans. on Computer Systems,* **9**(3), 272–314 (August 1991).
9. J. M. Chang and N. F. Maxemchuk, 'Reliable broadcast protocols', *ACM Trans. on Computer Systems,* **2**(3), 251–273 (August 1984).
10. M. F. Kaasshoek and A. S. Tanenbaum, 'Group communication in the AMOEBA distributed operating system', *Proc. 11th Int. Conf. on Distrib. Syst*., 1991, pp. 222–230.
11. S. Mishra, L. L. Peterson, and R. Schlichting, 'Experience with Modularity in Consul', *Software–Practice and Experience,* **23**(10), 1059–1075 (October 1993).
12. P. M. Melliar-Smith, L. E. Moser and V. Agrawala, 'Broadcast protocol for distributed systems', *IEEE Trans. on Parallel and Distributed Syst.,* **1**(1), 17–25 (January 1990).
13. B. Rajagopalan and P. K. McKinley, 'A token-based protocol for reliable, ordered multicast communication', *Proc. 8th IEEE Symposium on Reliable Distributed Systems*, Seattle, October 1989, pp. 84–93.
14. P. Verissimo, L. Rodrigues and M. Baptista, 'AMp: A highly parallel atomic multicast protocol', *ACM SIGCOMM Symposium*, 1989, pp. 83–93.
15. E. Nett and B. Weiler, 'Nested dynamic actions- how to solve the fault containment problem in a cooperative action model', *Proc. of 13th Symp. on Reliable Distributed Systems*, October 1994, pp. 106–115.
16. N. C. Hutchinson and L. L. Peterson, 'The x-Kernel: An architecture for implementing network protocols', *IEEE Trans. on Software Eng.,* **17**(1), 64–76 (January 1991).
17. M. Hiltunen and R. Schlichting, 'An approach to constructing modular fault-tolerant protocol', *Proc. of 12th Symposium on Reliable Distributed Systems*, Princeton, NJ, October 1993, pp. 112–121.
18. R. van Renesse and K. P. Birman, 'Horus: A flexible group communication system', *Comm. ACM,* **39**(4), 76–83 (April 1996).
19. L. Lamport, 'Time, clocks, and the ordering of events in a distributed systems', *Comm. ACM,* **21**(7), 558–565 (July 1978).
20. S. J. Leffler, M. K. Mckusick and J. S. Quarterman, *The Design and Implementation of 4.3BSA UNIX Operating System*, Addison-Wesley, 1989.
21. M. F. Kaasshoek and A. S. Tanenbaum, 'An evalution of the AMOEBA group communication system', *Proc. 16th Int. Conf. on Distrib. Syst*., Hong Kong, May 1996, pp. 436–447.
22. L. Moser, P. Melliar-Smith, A. Agrawala, R. Budhia, C. Lingley-Ppadopoulos and T. Archambault, 'The Totem System', *Proc. 25th Symposium on Fault-tolerant Computing Systems*, June 1995, pp. 61–66.
23. L. Rodrigues and P. Verissimo, 'xAMp: A multi-primitive group communication service', *Proc. of 11th IEEE Symposium on Reliable Distributed Systems*, Houston, TX, October 1992, pp. 112–121.
24. S. Luan and V. D. Gligor, 'A fault-tolerant protocol for atomic broadcast', *IEEE Trans. on Parallel and Distributed Syst.,* **1**(3), 271–285 (July 1990).
25. H. Garcia-Molina and A. Spauster, 'Ordered and reliable multicast communication', *ACM Trans. on Computer Systems,* **9**(3), 242–271 (August 1991).
26. S. Navaratnam, S. Chanson and G. Neufeld, 'Reliable group communication in distributed systems', *Proc. 8th Int. Conf. on Distributed Systems*, San Jose, CA, June 1988, pp. 439–446.
27. Y. Amir, D. Dolev, S. Kramer and D. Maiki, 'Transis: A communication sub-system for high availability', *22nd FTCS*, Boston, July 8–10 1992, pp. 76–82.
28. L. L. Peterson, N. Buchholz and R. Schlichting, 'Preserving and using context information in interprocess communication', *ACM Trans. on Computer Systems,* **7**(3), 217–246 (August 1989).