QOS PRESERVING TOTALLY ORDERED MULTICAST

ZIV BAR-JOSEPH, IDIT KEIDAR, TAL ANKER, NANCY LYNCH

ABSTRACT. This paper presents an algorithm for totally ordered multicast which preserves Quality of Service (QoS) guarantees. The paper assumes a QoS reservation model in which the network allows for reservation of variable bandwidth, specified by the average transmission rate and the maximum burst. As long as the application sends at the reserved rate, the network guarantees to deliver messages with bounded delays. For this model, the paper presents a totally ordered multicast algorithm that preserves the bandwidth and latency reserved by the application within certain additive constants that do not depend on the number of processes participating. This is an improvement over previous work, which gave latency bounds proportional to the number of processes. Furthermore, the presented algorithm allows for dynamic joining and leaving of processes while still preserving the QoS guarantees.

Keywords: Quality of Service (QoS), multicast, total order.

1. INTRODUCTION

Totally ordered multicast allows multiple processes to send messages, so that all the processes deliver messages in the same order. Totally ordered multicast is a useful paradigm for applications that replicate state using the state machine approach [Lam78, Sch90]. Much work has been dedicated to totally ordered multicast algorithms (e.g., [Lam78, CHD98, CM84]).

In the past few years, we have witnessed new applications that require *quality of service (QoS)* guarantees from the network (e.g., [McC92]). Some need strict guarantees on available bandwidth, others need a bound on the latency a packet can suffer when transmitted over the network.

Supported by by Air Force Aerospace Research (OSR) grants F49620-00-1-0097 and F49620-00-1-0327, Nippon Telegraph and Telephone (NTT) grant MIT9904-12, and NSF grants CCR-9909114 and EIA-9901592.

ATM networks [ATM96] allow applications to reserve QoS parameters such as bounded latency, guaranteed bandwidth and bounded loss rate. The *IETF Integrated Services* working group is concerned with adding similar QoS support to the Internet. The QoS parameters that the new services will support include, among others, bounded latency, guaranteed bandwidth reservation and bounds on message loss (see [SPG97]).

There are several applications that replicate some state with a certain degree of consistency and yet also require predictable message delays. Such applications can benefit from totally ordered multicast, as long as the introduction of total order does not introduce excessive delays. An example of such an application is a military command and control application, where several geographically distributed parties concurrently update the battlefield state. The shared state reflects information such as the current location of forces, the ammunition supply, intelligence information about the enemy whereabouts, strategic plans, etc. For such an application, timely updates are crucial. In addition, the involved parties need to observe a consistent state, for example, to agree upon the strategic plans. Other example applications include joint editing of a shared white-board [McC92], a shared text editor [UCL], and multi-user online strategy games [GD98, ten].

Applications such as those described above seldom exploit totally ordered multicast. This is because achieving total order requires delaying messages until agreement upon their order is reached, and many believe that this delay is too large. In this paper we show that totally ordered multicast can be achieved with guaranteed bounded delays. We present the first – as far as we know – totally ordered multicast algorithm which achieves delays that are *not* proportional to the number of processes participating in the multicast.

Specifically, we consider a network that provides QoS guarantees, and build on top of it a totally ordered multicast service which preserves the bandwidth and latency reserved by the application within some additive constants. We consider an underlying network with a reservation service for *Variable Bit-Rate (VBR)* [ATM96], which allows for some bursty transmission. We assume that while messages are sent at the reserved rate, they arrive within a bounded latency. We further assume that the network can provide a bounded loss rate.

Our algorithm tolerates process failures and recoveries. It allows for dynamic joining and leaving of processes *while still preserving the QoS* *guarantees*. This is in contrast to totally ordered group communication algorithms (e.g., [FvR97, WMK95, CHD98]) which typically block message delivery at re-configuration times, and thus induce a large latency.

1.1. **Totally ordered multicast versus Atomic Broadcast.** Totally ordered multicast guarantees that if two correct processes deliver the same two messages, then they deliver these messages in the same order. (The formal definition appears in Section 3). Unlike Atomic Broadcast [HT93], totally ordered multicast does not guarantee that correct processes deliver the same messages: if a faulty process sends a message, the message may be delivered by one correct process and not by another.

In [BJKAL00] we show that if processes can fail, an algorithm implementing Atomic Broadcast can guarantee, at best, a latency bound which is proportional to the number of failures it can tolerate. In this paper, on the other hand, we present an algorithm for totally ordered multicast, which provides constant latency bounds. These bounds do not depend on the number of failures or on the number of participating processes in any way.

We believe a totally ordered multicast service to be useful for applications that require real-time updates of a shared state. Consider, for example, the military application described above, where it is important for users to observe state changes in a timely manner. It is also desirable that all participants observe the state consistently. However, consistency cannot be achieved at the cost of compromising timeliness.

Using our totally ordered multicast service, updates are always timely, and are also consistent while no failures occur. Furthermore, if there is a point after which no failures occur, then after this point updates are always consistent, i.e., the same messages are delivered to all processes. When a process fails, the totally ordered multicast service can deliver an update that was initiated by the faulty process to one correct process and not to another. This can cause the state of two correct processes to differ. Such inconsistencies can be detected and reconciled, for example, using the algorithms of [KD96, FV97], which implement Atomic Broadcast atop group communication systems providing totally ordered multicast.

Several group communication systems provide totally ordered multicast; most do not address QoS issues. The only exception that we are aware of is RTCAST [ASJS96]. RTCAST assumes a stronger failure model than we do in this paper, and does not work with our failure model. Furthermore, RTCAST achieves a latency bound which is *linear* in the number of processes, and not constant as we do (cf. Section 7).

2. Model

We assume a static universe of n processes, with distinct identifiers in $\{1, \cdots, n\}$. Processes communicate by exchanging multicast messages within a *multicast group*[1]. Processes can voluntarily join and leave the multicast group at any time. Processes can fail by crashing and may later recover. We do not consider Byzantine (malicious) failures.

Processes use an underlying network communication service which allows for QoS reservation. Specifically, the network allows for reservation of variable bandwidth, specified by the average transmission rate and the maximum burst. As long as messages are sent at the reserved rate, the network guarantees to deliver messages with a bounded delay.

We present an algorithm that guarantees total ordering of messages and also preserves QoS. The algorithm is implemented by a *Totally Ordered Multicast (TO)* layer that resides between the application and the underlying network, as depicted in Figure 1.

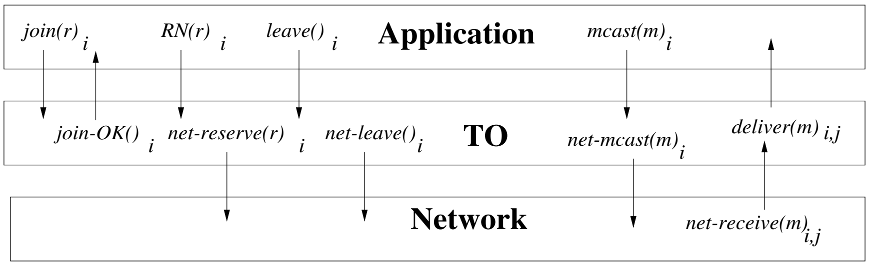

FIGURE 1. The TO service interface.

In Section 3 we specify the semantics of the TO layer. In Section 4 we specify our assumptions about the underlying network. We assume that processes are equipped with clocks, which are synchronized within a constant. We elaborate on this assumption in Section 5.

[1]For simplicity, we assume that a single multicast group exists.

3. Totally Ordered Multicast Service Specification

We now specify the totally ordered multicast service. This service is composed of the TO layer and the underlying network (cf. Figure 1). In this section we use the term *process* to refer to an application process running at a certain location. Processes use the service to send messages of a bounded size to the multicast group; the service delivers messages to all the processes in the same order.

Upon joining the multicast group, a process *reserves* the bandwidth required for its communication, that is, the process asks the service to allocate a certain bandwidth. If a process subsequently wishes to change its reserved bandwidth, it *renegotiates* its reservation parameters according to its new transmission rate.

Our service works within the framework of the VBR reservation model [ATM96], which allows applications to send bursty traffic. In this model, processes reserve an average transmission rate as well as a maximum burst size. Typically, the application declares its transmission rate in bytes per second. For simplicity, we assume that the rate is declared in units of messages per second. Since message size is bounded, these rates correspond closely.

Message sending is divided into time slots of a fixed length, Θ. Θ is the same for all the processes and is fixed throughout the execution. In addition, there exists a constant C which is the number of slots over which the average is computed. The rate is declared as two parameters:

1. *AppAvgRate* – the average message rate per Θ time. This means that $C * \mathit{AppAvgRate}$ is the maximum number of messages that may be sent during $C * \Theta$ time.
2. $AppMaxBurst$ – the maximum number of messages that may be sent during Θ time.

The application interface of the service is as follows:

- $join(r)_i$ is used by process i to join the multicast group and to reserve QoS. The structure r has two fields: *AppAvgRate* and $AppMaxBurst$, as explained above. This action is called initially, to establish the transmission rate before any messages are sent.
- $join\text{-}OK_i$ reports to process i that its latest join operation was successful, and i can now start sending messages.
- $leave_i$ is used by process i to leave the multicast group.

- $RN(r)_i$ is used by process i to *renegotiate* the QoS reserved from the network. The structure of r is as in the *join* action.
- $mcast(m)_i$ is used by process i to multicast message m to the multicast group. We assume that messages are unique, that is, the same message is not sent more than once. In addition, the messages are of a bounded size.
- $deliver(m)_{i,j}$ is used to deliver to process i a multicast message m that was previously multicast by process j.

We say that a message m is sent by a process i when $mcast(m)_i$ occurs, and that i delivers m when the totally ordered multicast service at process i performs the $deliver(m)_{i,j}$ action.

Definition 1. *If a process issues at least one* join *in an execution, and its latest* join *is followed by a* join-OK *but not by a* leave *or a failure, then the process is* correct *from the point of its* join-OK *onward.*

We assume that a process does not send messages until it joins the group and receives a *join-OK*, and does not send messages after leaving the group or recovering from a failure without re-joining. Furthermore, we assume that the application never exceeds the reserved sending rate.

When composed with an application that satisfies the assumptions above, the service satisfies the following property:

Definition 2. *Totally Ordered Multicast:*

- Total Order: *If processes i and j both deliver the same two messages m and m', they deliver these messages in the same order.*
- Integrity: *A message m is only delivered if it was previously sent, and is not delivered to the same process more than once.*
- Liveness: *If there is some point in the execution after which two processes i and j are correct, then every message that i sends after this point is eventually delivered by j.*

In addition to meeting the specification above, the algorithm is required to deliver messages within a bounded latency, *AppLatency*. This is the supremum over all executions, all messages m and all processes i of the time since the $mcast(m)_i$ action is performed in some execution until m is delivered by all processes that deliver it.

The latency bound supplements our Liveness property. The Liveness definition requires processes that remain in the group indefinitely from some point onward to eventually deliver each other's messages. The latency bound requires processes that do deliver a message to deliver it within a certain time bound. Since the algorithm cannot know which processes will remain live forever, it has to attempt to deliver messages within this time bound to all processes that are correct during this time.

4. The underlying network

We now describe our assumptions about the underlying network. In this section, we use the term *process* to refer to an instance of the TO layer running at a certain location.

The network preserves the FIFO order on messages sent between every pair of processes[2]. The network does not duplicate, corrupt, or spontaneously generate messages. In addition, we assume that the underlying network can guarantee a bounded loss rate (cf. Section 4.1 below).

In this paper we are only interested in studying cases in which QoS reservation and renegotiation are successful. Thus, for simplicity, we assume that all reservation requests made by a process are accepted by the network. Typically, QoS reservation and renegotiation take some time for the network to process. However, this time does not affect the message latency and for the sake of the analysis in this paper it is safe to ignore it. Therefore, for simplicity, we assume that once a reservation request is made, the bandwidth that was requested is immediately available to the reserving process.

We assume that the network supports the VBR reservation model. The network interface consists of the following types of actions:

- *net-reserve*$(r)_i$ is used by process i to join the multicast group and to reserve QoS from the network. The structure r has two fields: $NetAvgRate$ and $NetMaxBurst$. These are dual to the respective application QoS parameters described in the previous section. This action is called initially, to establish the transmission rate before any messages are sent. The action can be subsequently called to *renegotiate* the QoS reserved from the network.
- *net-leave*$_i$ is used by process i to leave the multicast group.
- *net-mcast*$(m)_i$ and *net-receive*$(m)_{i,j}$ are used by process i to multicast and receive messages from the network.

[2]Although messages sent over the Internet can rarely arrive out of FIFO order, this is easy to fix using sequence numbers.

Definition 3. *A process is considered a* member *of the group from the point it joins the group (using net-reserve$(r)_i$) onward, if it does not subsequently leave the group or fail.*

While the application sends at the reserved rate, we assume that the network guarantees a maximum message latency Δ, and a bounded loss rate. That is, the QoS reservation service is used to reserve this latency and loss rate. These parameters are the same for all processes (unlike the transmission rate which is unique to each process). For simplicity, we assume that the same latency Δ and loss rate are successfully reserved by all processes. Thus, any message sent from some process i using the *net-mcast*$(m)_i$ action will reach every process j via the *net-receive*$(m)_{j,i}$ action in at most Δ time. In the next section we explain the nature of the loss rate parameter and how its being used.

4.1. **Message loss.** In the literature, several different ways to express a QoS parameter bounding the loss rate were suggested. We consider the following definition when discussing our loss rate parameters:

Burst loss sensitivity [Par92] defines the maximum number of messages that can be lost in a given interval. The application specifies a *loss interval*, $x = k + l$, in terms of a number of consecutive messages from the same sender. The application then specifies a bound, l, on the number of messages sent in the same interval that the network can lose[3]. Formally:

Definition 4. *Burst loss sensitivity of l out of $k + l$ guarantees that if there is some point in the execution after which two processes i and j are group members, then for every $k + l$ consecutive messages that i sends after this point, j receives at least k.*

5. Clock synchronization and scheduling

We assume each process i has an internal clock denoted by $clock_i$. We assume that the difference between $clock_i$ and the real time is bounded. We denote by *now* the real time that has passed from the beginning of the execution[4] (thus, each execution starts with $now = 0$). We assume that there exists a constant Γ such that the maximum difference between

[3]Burst loss sensitivity enumerates the maximum number of messages of MTU (maximum transmission unit) size or smaller that may be lost. We assume that messages are small enough to benefit from this guarantee.

[4]The real time is used as an abstraction for the latency analysis.

$clock_i$ and now is at most $\Gamma/2$: for each process i: $now - \Gamma/2 \leq clock_i \leq now + \Gamma/2$.

This implies that the maximum difference between two processes' internal clocks is at most Γ. The assumption that clocks are synchronized within a bound is very reasonable. For example, the *Network Time Protocol (NTP)* [Mil92] can synchronize clocks to within one to fifty milliseconds on most network environments today. The synchronization level depends on the network technology, and on the distances between the synchronizing processes.

In addition to clock synchronization, we make the assumption that each process can precisely schedule events according to its local clock.

6. The Totally Ordered Multicast Algorithm

We now present our totally ordered multicast algorithm. The algorithm is *symmetric*, in the sense that all the participating processes play identical roles in the algorithm. The algorithm is composed of two layers: a *forward error correction (FEC)* layer which overcomes message loss, and an Ordering layer, as depicted in Figure 2.

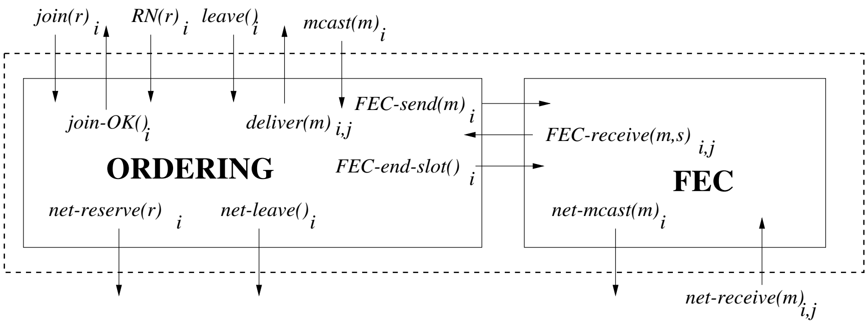

Figure 2. The TO service decomposition.

The algorithm divides the time into slots of length Θ. Processes use slots for sending and delivering messages. Messages are sent and delivered via the FEC layer, using the *FEC-send* and *FEC-receive* actions. The algorithm ensures that an application message m, and all the redundant FEC information associated with m, are sent (using *net-mcast*) in the same slot as m was *mcast* by the application. To this end, the FEC layer needs to be informed when a slot ends. This is done using the *FEC-end-slot* action.

The FEC layer is described in Section 6.1. In Section 6.2, we describe how the Ordering algorithm works while the network situation is static (that is, while there are no process failures, and processes do not dynamically join and leave the multicast group). In Section 6.3, we explain how the algorithm deals with process failures and dynamic joining and leaving. In the Appendix, we present pseudo-code for the complete Ordering algorithm as a timed automaton.

6.1. **Forward Error Correction.** FEC is a common technique to overcome bounded message loss by sending redundant messages. We use FEC to eliminate loss among correct processes: *net-mcast*$(m)_i$ and *net-receive*$(m, s)_{i,j}$ actions are executed by the FEC layer, and all lost messages are recovered by this layer.

Assuming burst loss sensitivity of l out of $k+l$, the FEC layer constructs l redundant messages per each k messages sent using *FEC-mcast* (by the Ordering layer). We refer to such a set of k original messages plus l redundant messages as a *FEC block*. The original k messages can be reconstructed from *any* k messages of the FEC block. FEC techniques for creating such redundant messages are well-studied in the literature (e.g., in [NBT97]), and we do not discuss them here.

The FEC layer organizes messages according to slots. The Ordering layer uses the *FEC-end-slot* action to inform the FEC layer when a slot ends. The FEC layer uses this action to keep track of which slot it is in. The slot number is attached to each message. When a slot ends, the FEC layer sends l redundant messages for the remaining messages in the terminating slot. Thus, the last FEC block of a slot may contain less than $k+l$ messages, in case the number of messages sent in the slot is not a multiple of k.

On the receiving end, the FEC layer delivers messages via *FEC-receive*(m, s), where s is m's sending slot number. FEC notifies the Ordering layer when all the messages from a certain sender for a slot have been received by delivering a special $\perp$ message.

We now explain the FEC layer operation. When *FEC-mcast*(m) occurs, the FEC layer holds on to message m. When the FEC layer has $k+1$ messages, it constructs a new FEC block with the first k plus l redundant messages. It tags each message in the block with its slot number, its number in the block (from one to $k+l$), and with the number of messages in the FEC block ($k + l$ in this case). The latter two numbers are used to reconstruct the messages at the receiving end. It then multicasts all $k + l$ messages.

When FEC-*end-slot* occurs, the FEC layer constructs a FEC block with l redundant messages for the remainder of the original messages for this slot. It tags the messages with a special *end* bit. So if the FEC has c unsent messages for slot s when slot s ended ($c \leq k$), it will now send $c + l$ messages. The number of messages in the FEC block, $c + l$, is included in each message header.

Processing on the receiving end is done in FEC blocks. If original messages of a FEC block arrive without gaps, the messages are immediately delivered. If there are gaps, that is, if a message is lost, then the layer waits to receive enough messages from this block – c from a block of $c + l$ messages, and then reconstructs the missing messages. If the layer receives more messages from this block it discards them. By the time the entire FEC block arrives, all the messages can be reconstructed and delivered. Messages are delivered to the Ordering layer using FEC-*receive* along with their sending slot number, which was included in the message header.

The FEC layer at process i knows that it processed the last block from sender j for slot s if it delivers messages that have an *end* bit. After processing this last block, it delivers a special $\perp$ message to the Ordering layer using FEC-*receive*$(\perp)_{j,i}$.

If the application does not send any message in slot s, the FEC layer generates a dummy message and l redundant messages for it. Upon receipt, the dummy message is discarded, and a $\perp$ message is delivered.

6.2. **The Ordering Algorithm: Static Case.** We now explain how the Ordering algorithm works while the network situation is static. Upon initiating the algorithm, the application specifies its transmission rate in the *join* action. The Ordering algorithm, in turn, reserves the required rate from the network. We later discuss the rate that needs to be allocated, as a function of the application's actual transmission rate and the message loss rate.

Sending and receiving of messages is done through the FEC layer. When a slot ends (that is, Θ time has passed from the time it started) the Ordering layer notifies the FEC layer.

In each slot, the algorithm delivers messages according to the process indices, i.e., it delivers all the messages for this slot sent by process 1, then all the messages sent by 2 etc. If process i is currently delivering messages for slot s from process j, i will move to deliver the messages from $j+1 \bmod n$ after it receives a $FEC\text{-}receive(\perp)_{j,i}$ message from the FEC layer.

We now discuss the correctness, and the QoS guarantees of our algorithm when no processes fail, join or leave.

Correctness. In order to prove correctness, we have to show that Definition 2 holds. Integrity is trivially satisfied from our assumptions on the network. Since we have a guarantee for a bounded loss rate from the network, and we use the FEC layer, all processes will be able to reconstruct all the original messages sent for each slot by other processes. Total Order and Liveness hold since for every slot s, all processes deliver all the messages sent for s in the same order. This is guaranteed by the fact that j will not move on to deliver messages from $i+1$ for slot s before it received a notification from the FEC layer that it had delivered the last message from i for slot s. Since the last FEC block for each slot (which is always sent when the TO layer performs the $FEC\text{-}end\text{-}slot$ action) carries with it the *end* flag, the FEC layer at j will know when to notify the TO layer that it have received all the messages sent by i for slot s, and thus the total ordering is guaranteed.

Latency. The maximum delay caused by this algorithm is: $AppLatency = \Delta + \Gamma + \Theta$. This is due to the fact that we deliver all messages sent by each process for slot s at the same time. Thus, a message sent by the application at process i at the beginning of s may be delivered after messages sent by the application at j at the end of s (see Figure 3).

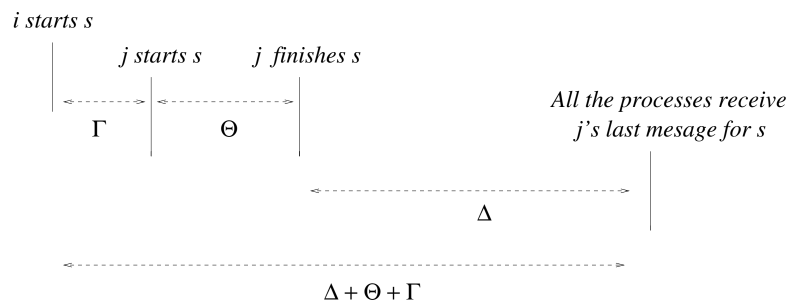

FIGURE 3. Maximum delay of the TO algorithm, with no failures.

Rate. The messages this algorithm adds over the messages sent by the application are the redundant messages added by the FEC layer. In addition, in case no messages are sent in a slot, a dummy message is added by the FEC layer. The latter effects the average rate, but not the maximum rate. Thus, upper bounds on the average and maximum rates used by our algorithm are:

$$NetAvgRate = AppAvgRate + (\lceil AppAvgRate/k \rceil + 1) * l + 1$$

$$NetMaxBurst = AppMaxBurst + (\lceil AppMaxBurst/k \rceil + 1) * l$$

6.3. **Process Failures, Joins and Leaves.** In this section we explain how our algorithm handles process failures, joins, and leaves.

Process failures. Recall that our definition of totally ordered multicast does not require correct processes to agree upon messages delivered by faulty ones. Therefore, the algorithm can deliver messages at each process without checking if other processes received these messages as well. In order to avoid waiting for messages from faulty processes, we implement an internal failure detector at each process.

According to the network guarantees, if a process sends a message all other processes receive it in at most Δ time. Therefore, if a process i waits more than $\Delta+\Gamma$ time from the time it finished slot s for a message from process j for this slot, then i knows that j has failed. Once process i detects that process j failed, it stops waiting for messages from j, and delivers the rest of the messages sent by other processes for slot s. From slot $s+1$ until j is added again (as we describe below), i does not deliver messages from j.

Dynamic Joins and Leaves. For a joining process j we allow an *initialization time* during which j does not have to deliver all messages it receives (it is not considered correct for the sake of the definition in Section 3), and cannot send messages. At the end of this initialization period, a $join\text{-}OK_j$ notifies j that it may start sending.

When a process j wants to join the algorithm, it needs to know in which slot the rest of the processes are. To this end, j checks its own clock, and computes the slot the execution has reached. Since each execution starts with $now = 0$, j can know the slot it is in by setting s to $clock_j/\Theta$. In order to join, j sends a $(\text{“}join\text{”}, s)_j$ message to all processes where s is the slot in which j should be added. Whenever a process receives a "join" message it immediately processes it, and records that j should be added at slot s. Since this message arrives at all processes at most Δ time after it is sent, and since the difference between j's clock and all other processes' internal clocks is at most Γ, j can know that all the processes will receive the message before they start slot $Smax \stackrel{\text{def}}{=} (clock_j + \Delta + \Gamma)/\Theta + 1$. Thus, j sends $(\text{"}join\text{"}, Smax)_j$ to all processes, and starts sending and delivering messages from slot $Smax$. During slot $Smax$, j assumes all processes to be correct, that is, tries to deliver messages from them unless it detects a failure by time-out as explained above.

When a process i receives j's $(\text{"}join\text{"}, Smax)_j$ message, it adds j to its list of active processes upon starting to deliver messages for slot $Smax$. From $Smax$ onward (or until j is detected as failed again), i waits for messages from j in every slot.

Since all the active processes start slot $Smax$ after j wakes up, j will receive all the messages sent for slot $Smax$. Furthermore, all the processes will receive j's *join* message before they start this slot. Process j only joins the algorithm (that is, starts delivering messages to the application) at slot $Smax$, and it discards all messages it receives for earlier slots. Note that j is added to the algorithm at the same slot ($Smax$) by all the processes (including j itself). Process j starts this slot at the latest $\Delta + \Gamma + \Theta$ time after it wakes up.

The application leaves the algorithm using the *leave* action. When $leave()_j$ is performed, the Ordering layer at process j notifies all other processes by sending a ("*leave*") message. It then waits to the end of this sending slot, performs the *net-leave* action, and exits the algorithm. This makes sure that the "leave" message is sent by the FEC layer. Upon receipt, a "leave" message is stored in the message queue. When it is time for process i to deliver a message from process j, and the next message from j in the queue is a "leave", process i stops waiting for messages from j: it continues to deliver messages sent by other processes and does not deliver messages from j until j is added again.

Correctness. Integrity follows directly from the correctness proof in the previous section. We now show that Total Order is guaranteed: Failures can cause messages delivered by one process not to be delivered by another process, when the sending process has failed. However, the order of message delivery (by slot, and in each slot according to processes indices) remains the same and the delivery order for messages that are delivered is preserved. When a process j joins, all processes receive j's join message with j's join slot before they begin to deliver any application messages from j. As argued above, this message reaches all processes before j's joins slot, and all processes add j in this slot.

We now prove Liveness. If two processes i and j are correct from some point onward, then they are also both members of the group from this point onward. Therefore, by the guarantees of the network and the FEC layer, every message that is sent by i after this point is received by j. If a process j is correct in slot s, and j sends a message m in this slot, then every other correct process receives j's "join" message before m. Furthermore, since j is correct, i does not time-out on j in slot s (as explained above). Therefore, i delivers m.

Latency. In this algorithm, a process j waits at most $\Delta + \Gamma$ time after a slot s ends before j times-out on faulty processes and delivers all the received messages for the slot. Thus, j delivers all the messages for a slot s at most $\Delta + \Gamma + \Theta$ time after it starts slot s. Furthermore, j starts s at most Γ time after every other process i starts slot s (see Figure 4). Therefore, j will deliver all messages sent for s at most $\Delta + \Gamma + \Theta + \Gamma = \Delta + 2\Gamma + \Theta$ time from the time it was sent by the application in i, and: $AppLatency = \Delta + \Theta + 2\Gamma$.

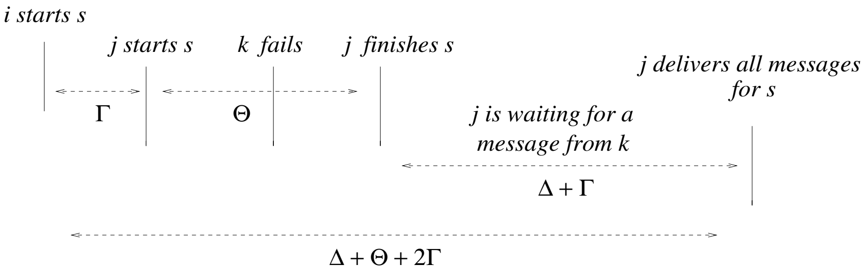

FIGURE 4. Maximal delay caused when process failures are tolerated.

Rate. The transmission rate is not affected by the added support for fault tolerance. Both $NetAvgRate$ and $NetMaxBurst$ remain the same as in the previous section, since no new messages are added to the algorithm presented in this section with the exception of "join" messages. We do not take the cost of initialization and renegotiation into account when analyzing the cost of the "normal" operation of the algorithm.

7. Related Work

The only previous work that we are aware of addressing QoS guarantees of totally ordered multicast primitives is RTCAST [ASJS96]. The failure model assumed in RTCAST is weaker than the one we assume. There, it is assumed that if a process p fails, and a correct process q receives, from the network, some message m sent by q before its failure, then every other correct process will receive m as well. In contrast, we allow the network to deliver a message from a faulty process to some correct process and not to another. In the failure model of RTCAST, our algorithm, as is, solves Atomic Broadcast. The latency bound achieved by RTCAST is *linear* in the number of processes, and not constant as we do here.

8. Conclusions and Future Work

This paper sets the framework for analyzing QoS guarantees of multicast primitives. Specifically, we have presented an algorithm for totally ordered multicast that does not violate QoS guarantees. We considered the variable bit rate model, where processes can send bursty traffic. We have shown that our totally ordered multicast algorithm preserves the latency of the underlying network within a constant, and this latency does not depend on the number of processes or the number of failures.

Totally ordered multicast is a weaker service than Atomic Broadcast. With totally ordered multicast, it is possible for one correct process to deliver a message from a faulty process while another correct process does not deliver this message. In [BJKAL00] we show that if processes can fail, an algorithm implementing Atomic Broadcast can guarantee, at best, a latency bound which is proportional to the number of failures it can tolerate. Thus, there is a tradeoff between the consistency of Atomic Broadcast and the timeliness of totally ordered multicast.

Understanding of this tradeoff may suggest a certain design for applications, like military command and control, that need to present updates in a timely manner even if they are not entirely consistent, as long as they later converge to a consistent state. Such applications may exploit algorithms such as the one suggested here for fast updates, and in the background implement Atomic Broadcast order so that the data will eventually be consistent.

This paper is a first step in the way of providing a mathematical approach to understanding the costs of different services in different network models. Such understanding is vital for efficient design of distributed applications. Future work will consider QoS preserving Atomic Broadcast. Although constant bounds cannot be achieved for this problem, it is interesting to study the bounds that can be achieved. Future work will also consider richer network models, making the suggested algorithms more suitable for large scale distributed applications on the Internet, such as collaborative text editing. An interesting issue to consider is tolerating network partitions and reconnects; we discuss this issue in the full paper.

Other interesting future work can consider different QoS parameters. For example, instead of analyzing a fixed latency bound one may want to consider the average latency and the maximum jitter (that is, the variation of the latency). Randomized algorithms may be able to provide such QoS guarantees most effectively.

References

[ASJS96] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. RTCAST: Lightweight multicast for real-time process groups. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, June 1996.

[ATM96] The ATM Forum Technical Committee. *ATM User-Network Interface (UNI) Signalling Specification Version 4.0, af-sig-0061.000*, July 1996.

[BJKAL00] Ziv Bar-Joseph, Idit Keidar, Tal Anker, and Nancy Lynch. QoS Preserving Totally Ordered Multicast. Technical Report MIT-LCS-TR-796, MIT Laboratory for Computer Science, January 2000.

[CHD98] G. Chockler, N. Huleihel, and D. Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *17th ACM Symp. on Prin. of Distributed Computing (PODC)*, pages 237–246, June 1998.

[CM84] J. Chang and N. Maxemchunk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3), 1984.

[FV97] R. Friedman and A. Vaysburg. Fast replicated state machines over partitionable networks. In *16th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, October 1997.

[FvR97] Roy Friedman and Robbert van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *6th IEEE International Symp. on High Performance Distributed Computing*, 1997.

[GD98] Laurent Gautier and Christophe Diot. Design and Evaluation of MiMaze, a Multi-player Game on the Internet. In *Proceedings of IEEE Multimedia Systems*, June 28 - July 1 1998. URL http://www-sop.inria.fr/rodeo/MiMaze/.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *chapter in: Distributed Systems*. ACM Press, 1993.

[KD96] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *15th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 68–76, May 1996.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 78.

[McC92] S. McCanne. *A Distributed Whiteboard for Network Conferencing*. UC Berkeley CS Dept., May 1992. Unpublished Report. Software available from ftp://ftp.ee.lbl.gov/conferencing/wb.

[Mil92] David L. Mills. *Network Time Protocol (Version 3) Specification, Implementation, RFC 1305*, March 1992. Internet Engineering Task Force, Network Working Group.

[NBT97] J. Nonnenmacher, E. Biersack, and D. Towsley. Parity-based loss recovery for reliable multicast transmission. In *ACM SIGCOMM*, 1997.

[Par92] C. Partridge. *A Proposed Flow Specification, RFC 1363*, September 1992. Internet Engineering Task Force, Network Working Group.

[Sch90] F. B. Schneider. Implementing fault tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SPG97] S. Shenker, C. Partridge, and R. Guerin. *Specification of Guaranteed Quality of Service, RFC 2212*, September 1997. Internet Engineering Task Force, Network Working Group.

[ten] *10Six - A massive online team play over the Internet.* URL: http://www.tensix.com/ , http://www.heat.net/10sixchannel/index.html.

[UCL] The UCL Networked Multimedia Research Group. *NTE: Network Text Editor.* URL: http://www-mice.cs.ucl.ac.uk/multimedia/software/nte/.

[WMK95] B. Whetten, T. Montgomery, and S. Kaplan. A high perfomance totally ordered multicast protocol. In K. P. Birman, F. Mattern, and A. Schipper, editors, *Theory and Practice in Distributed Systems: International Workshop*, pages 33–57. Springer Verlag, 1995. LNCS 938.

Appendix A. Timed Automaton Implementing the Ordering Algorithm

We now present a timed I/O automaton implementing the Ordering algorithm, for each process i running the algorithm. Figure 5 presents the signature and state variables of the automaton. Figures 6 and 7 present the transitions.

Authors addresses:

Z. Bar-Joseph, MIT Lab for Computer Science, zivbj@mit.edu

I. Keidar, MIT Lab for Computer Science, idish@theory.lcs.mit.edu

T. Anker, Hebrew University of Jerusalem, anker@cs.huji.ac.il

N. Lynch, MIT Lab for Computer Science, lynch@theory.lcs.mit.edu

Signature

Input:
 $join(r)_i$, r a structure with two integer fields
 $RN(r)_i$, r a structure with two integer fields
 $leave()_i$,
 $mcast(m)_i$, $m \in M$
 $FEC\text{-}receive(m,s)_{i,j}$, $m \in M$, $s \in N$
Output:
 $join\text{-}OK()_i$
 $FEC\text{-}end\text{-}slot()_i$
 $FEC\text{-}send(m)_i$ $m \in M$
 $deliver(m)_{i,j}$, $m \in M$
 $net\text{-}reserve(r)_i$ r a structure with two integer fields
 $net\text{-}leave()_i$
Time-passing:
 $v(t), t \in R^+$
Internal:
 end-deliver
 failure-detector
 skip-failed
 process-leave
 end-recvSlot
 wait-start

State

For all j, $Rqueue(j)$, a FIFO queue of messages, initially empty
$Squeue$, a FIFO queue of messages, initially empty
$current$, an integer initially 1, // current process to receive from
$myJoin$, an integer initially 0
$finished$ unbounded array of reals, initially 0 in all places
$Join$ array of size n of integers, initially 0 in all places
$Failed$ a group of process indices, initially empty
$sendingSlot, recvSlot$, integers initially 1,
$rate$, pair of integers initially $\perp$
$changeRate, Pleave, Nleave$, boolean initially FALSE
$clock \in R^{\geq 0}$, initially 0
$last \in R^+ \cup \{\infty\}$, initially ∞

FIGURE 5. The Ordering automaton for process i: signature and state.

Transitions

Input $join(r)_i$
Eff: $rate = r$
$changeRate = TRUE$
$myJoin = (clock + \Delta + \Gamma)/\Theta + 1$
add $(''join'', myJoin)$ to $Squeue$
$recvSlot = myJoin$
$sendingSlot = \lceil clock/\Theta \rceil$
$last = (sendingSlot + 1)\Theta$

Output $join\text{-}OK()_i$
Pre: $sendingSlot = myJoin$
Eff:

Input $RN(r)_i$
Eff: $rate = r$
$changeRate =$ TRUE

Output $net\text{-}reserve(r)_i$
Pre: $clock = last$
$changeRate =$ TRUE
$r = rate$
Eff: $changeRate =$ FALSE

Input $leave()_i$
Eff: $Pleave =$ TRUE
add $(``leave'')$ to $Squeue$

Output $net\text{-}leave()_i$
Pre: $Nleave =$ TRUE
Eff:

Input $mcast(m)_i$
Eff: add m to $Squeue$

Output $FEC\text{-}send(m)_i$
Pre: (m) is first on $Squeue$
Eff: discard first element of $Squeue(j)$

Internal $wait\text{-}start$
Pre: $sendingSlot < myJoin$
$clock = last$
Eff: $last := clock + \Theta$
$sendingSlot++$

Output $FEC\text{-}end\text{-}slot()_i$
Pre: $sendingSlot \geq myJoin$
$clock = last$
$Nleave = FALSE$
$changeRate = FALSE$
$Squeue$ is empty
Eff: if($Pleave =$ TRUE)
$Nleave =$ TRUE
else {
$last := clock + \Theta$
$finished[sendingSlot] = clock$
$sendingSlot++$ }

FIGURE 6. The Ordering algorithm automaton for process i: transitions, part 1.

Transitions

TimePassage $v(t)$
  Cho: $p \geq 0$
  Pre: $now + t - \Gamma/2 \leq clock + p \leq now + t + \Gamma/2$
    $clock + p \leq last$
  Eff: $now := now + t$
    $clock := clock + p$

Input $FEC\text{-}receive(m, s)_{i,j}$
  Eff: if $(m = (''join``, s))$
      $Join[j] := s$
    elseif $(s \geq myJoin)$
      add (m) to $Rqueue(j)$

Output $deliver(m)_{i,j}$
  Pre: $current > 0$
    $j = current$
    (m) is first on $Rqueue(j)$
    $m \neq \perp$ && $m \neq'' leave''$
  Eff: discard first element of $Rqueue(j)$

Internal $process\text{-}leave$
  Pre: $current > 0$
    $j = current$
    (m) is first on $Rqueue(j)$
    $m ='' leave''$
  Eff: $Failed := Failed \cup j$
    discard all elements of $Rqueue(j)$
    $current = current + 1$

Internal $end\text{-}deliver$
  Pre: $current > 0$
    $j = current$
    $\perp$ is first on $Rqueue(j)$
  Eff: discard first element of $Rqueue(j)$
    $current := (current + 1) \bmod (n + 1)$

Internal $end\text{-}recvSlot$
  Pre: $current = 0$
  Eff: $current := 1$
    $recvSlot++$
    for all j s.t. $Join[j] = recvSlot$
      $Failed := Failed \setminus \{j\}$

Internal $failure\text{-}detector$
  Pre: $current > 0$
    $j = current$
    $clock \geq finished[recvSlot] + \Delta + \Gamma$
    $Rqueue(j)$ is empty
  Eff: add j to $Failed$
    $current := (current + 1) \bmod (n + 1)$

Internal $skip\text{-}failed$
  Pre: $current \in Failed$
  Eff: $current := (current + 1) \bmod (n + 1)$

FIGURE 7. The Ordering algorithm automaton for process i: transitions, part 2.