QoS Preserving Totally Ordered Multicast

Ziv Bar-Joseph
Lab for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
zivbj@mit.edu

Idit Keidar
Lab for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
idish@theory.lcs.mit.edu
http://theory.lcs.mit.edu/∼idish

Tal Anker
Computer Science Institute
The Hebrew University of Jerusalem
Jerusalem, 91904, Israel
anker@cs.huji.ac.il
http://www.cs.huji.ac.il/∼anker

Nancy Lynch
Lab for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
lynch@theory.lcs.mit.edu
http://theory.lcs.mit.edu/∼lynch

January 24, 2000

Abstract

This paper studies the Quality of Service (QoS) guarantees of totally ordered multicast algorithms. The paper shows that totally ordered multicast can coexist with guaranteed predictable delays in certain network models. The paper considers two reservation models: constant bit rate (CBR) and variable bit rate (VBR). For these models, the paper presents totally ordered multicast algorithms that preserve the bandwidth and latency reserved by the application within certain additive constants. Furthermore, the paper presents an algorithm that tolerates message loss (in which case, there can be gaps in the total order) and 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 of some sort using the state machine approach [23, 38].

Much work has been dedicated to totally ordered multicast algorithms. Roughly, such algorithms may be classified as symmetric [23, 11, 34, 7], where all the processes execute the same algorithm to independently determine the total order, or leader based [6], where one of the processes, (the leader), determines the total order and informs the others. A hybrid approach is presented in [37], where processes that transmit at a high rate use the symmetric approach and other processes use a leader based approach.

In the past few years, we have witnessed many new applications that involve multimedia multicast and unicast (e.g., [30, 29, 31]). These applications require *quality of service (QoS)* guarantees from the network; some need strict guarantees on available bandwidth, others need a bound on the latency a packet can suffer when transmitted over the network. ATM networks allow applications to reserve QoS parameters such as bounded latency and guaranteed bandwidth; they provide QoS classes such as *Constant Bit-Rate (CBR)* and *Variable Bit-Rate (VBR)* [5]. The *IETF Integrated Services* working group [19] 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 [43, 39]).

There are several applications that replicate 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 an online strategy game [15, 1] which allows multiple users to play at the same time and see the same image of the game. Bounded delay is important for the "real time" feeling of the game. Totally ordered multicast is important for keeping the view of the game consistent. For example, if one player shoots and another simultaneously moves away, it is important that both observe the shooting and moving as occurring in the same order because otherwise the moving player will be observed as dead by one player while he will be considered alive by another. Other example applications include joint editing of a shared white-board [30, 29], a shared text editor [41], and military command and control applications.

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. For example, in his book Internetworking Multimedia [10], Crowcroft writes:

> *"The requirements of resilience and scalability dictate that total consistency of view is not possible unless mechanisms requiring unacceptable delays are employed."*

The idea that consistency and predictable delays are mutually exclusive is at the root of design decisions made in building such applications [30, 41]. Such applications usually settle for weak consistency constraints and run application-specific algorithms to detect and resolve inconsistencies.

In this paper we study the tradeoffs between different levels of consistency and QoS guarantees. We show that totally ordered multicast can coexist with guaranteed predictable delays in certain network models. The network models we discuss are defined in Section 2. In Section 3 we define two different semantics of totally ordered multicast – reliable (gap free) total order (which is often called Atomic Broadcast [16]), and total order with gaps. Reliable total order can provide strong consistency. Total order with gaps can provide a weaker form of consistency. Total order with gaps may be extended to eventually provide strong consistency, under certain conditions on network behavior, e.g., by the algorithms of [21, 22, 2, 14, 12] that are not application-specific. We analyze the QoS guarantees that can be made by algorithms providing these two semantics in various different network models.

Previous papers (e.g., [13, 37]) measured and analyzed average delays of total order protocols. However, they did not assume QoS and did not prove any bounds on the delay. We are not aware of previous work studying QoS guarantees of totally ordered multicast.

For most of the paper, we consider a reservation service for two QoS parameters — bandwidth and latency, and we assume no message loss. In Section 7 we also consider a third QoS parameter – a bound on the number of consecutive messages that may be lost. We consider two types of bandwidth reservations, CBR and VBR. We present symmetric totally ordered multicast algorithms that preserve the reserved bandwidth and latency within some additive constants. Our algorithms are variations on the algorithm of [7], tailored to the specific network and reservation models.

In the CBR model, the application reserves a fixed bandwidth. The application can send at a lower rate than the reserved rate at certain times, but it cannot send bursts that exceed the reserved rate. In any case, the application is charged for the reserved rate regardless of the sending rate. In Section 4, we present an algorithm for reliable totally ordered multicast for this model in a failure-free environment. Our algorithm does not require additional bandwidth over the reserved rate with the exception of in-band messages sent at start-up and whenever QoS *renegotiation* occurs (i.e., when the application changes its reserved QoS parameters). Furthermore, our algorithm preserves a tight bound on message latency: if the processes' clocks are synchronized with each other with a deviation of at most Γ, then the totally ordered multicast algorithm delivers messages with a maximum latency of $\Delta + \Gamma$, where Δ is the maximum latency of the underlying network.

In the VBR model, the application reserves two parameters: the application's average transmission rate over long periods of time, and a maximum burst size, which is the maximum the application can send during a short period of time which we call a slot. The slot size is denoted Θ. In Section 5, we present an algorithm for reliable totally ordered multicast for this model in a failure-free environment. Our algorithm preserves the reserved maximum burst size and induces an overhead of at most $1/\Theta$ over the average transmission rate; in other words, the totally ordered multicast algorithm needs to reserve an average transmission rate that is larger by $1/\Theta$ than the average rate reserved by the application. The maximum latency guaranteed by this algorithm is $\Delta + \Gamma + \Theta$. Thus, with this algorithm, there is a tradeoff between the increase in the average transmission rate overhead and the increase in latency.

In Section 6 we extend our algorithm for the VBR model to allow for process failures and joins, and also comment on the handling of network partitions. In Section 7 we extend the model further to allow for message loss. We show that if processes can fail, an algorithm implementing reliable total order can guarantee, at best, a latency bound which is proportional to the number of failures it can tolerate. We therefore relax the requirements of the totally ordered multicast protocol to allow gaps in the total order when discussing failure prone cases. We then present an algorithm that tolerates message loss and 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., [17, 13, 42, 3, 33, 7]) which typically block message delivery at re-configuration times and thus induce a large latency at such times. When adding support for join/leave and message loss, the maximum latency of the VBR algorithm is increased by Γ to become $\Delta + 2\Gamma + \Theta$. The average transmission rate and maximum burst of the VBR algorithm are not affected by the added support for join/leave. The handling of message loss increases the transmission overhead by adding an integer to the header of every message. When a failed process recovers, it can join the algorithm $\Delta + \Gamma + \Theta$ time after it wakes up.

In Section 8 we analyze the QoS guarantees of leader-based algorithms. Handling fault tolerance in leader-based algorithms is generally difficult, and is bound to introduce long delays. Furthermore, we show that in the model of this paper, QoS guarantees of leader-based algorithms are inferior to those of symmetric ones. This motivates our focus on symmetric algorithms in this paper. Section 9 concludes the paper and discusses future directions.

2 Model

We assume a static universe of n processes Where we assume process failures, processes fail by crashing and may later recover.

We assume a message-passing model. Our model consists of three layers, as shown in Figure 1: the network layer, the totally ordered multicast layer (denoted *TO algorithm*), and the application layer. In this section we explain the assumptions we make regarding the network layer. Since the network layer guarantees depend on proper behavior of the application, we describe the expected behavior of the application in each of the network models. Several algorithms implementing the TO layer will be discussed in the following sections.

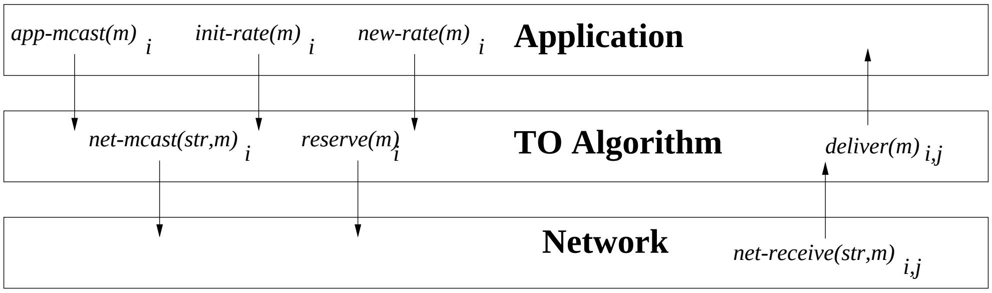


Figure 1: The network and TO service interfaces.

2.1 Network Model

Processes communicate by exchanging multicast messages. The message size is bounded. The network preserves a FIFO order on messages sent between every pair of processes[1]. The network does not duplicate, corrupt, or spontaneously generate messages.

We assume that the network allows for reservation of two QoS parameters: bandwidth and latency. We assume that the network guarantees a maximum message latency Δ (i.e., the reservation service is used to reserve this latency). The delay is the same for all processes: any message sent from some process i (the $net\text{-}mcast(str, m)_i$ action in Figure 1) will reach every process j (via the $net\text{-}receive(str, m)_{i,j}$ action) in at most Δ time. The network can be modeled as a timed automaton (see [27], Ch. 23).

In this paper we assume a sender based reservation protocol (as in the ATM model [4, 5]), although some of the common reservation protocols (e.g., [44]) are receiver based. In Section 4 we explain how our algorithms can work with receiver based reservation without additional delay.

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

Each process i reserves the bandwidth it needs, according to the application transmission rate. The reservation is done by the $reserve(m, \Delta)_i$ request made to the network ($reserve(m, \Theta, \Delta)_i$ for the VBR model). Each time a sending process changes its transmission rate, it must renegotiate its reservation with the network. Since we are only interested in studying cases in which the reservation and renegotiation are successful, we limit our attention to such cases. Thus, we assume 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, and a process may start sending messages at the reserved / renegotiated rate only after this time. 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 ignore this time in our analysis and assume that once a reservation request is made, the bandwidth that was requested is immediately available to the reserving process.

In this paper we look at two different reservation models: CBR and VBR. In both models, the application typically 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.

In Section 4 we assume the CBR reservation model. In this model, the reservation is requested from the network using the $reserve(m, \Delta)_i$ action. Δ is the maximum latency, and m is an integer that holds the message rate i wishes to reserve. We assume that Δ is fixed throughout this paper; therefore, for simplicity, we omit it and write $reserve(m)_i$. If the application reserves a rate $NetRate$ of m messages per second, it is then assumed to send at most one message each $1/m$ second (that is, it can either send or not send a message each $1/m$ second, but it cannot send more than one message over $1/m$ second). The application pays for the rate it reserves whether it uses it or not.

In Sections 5–8 we assume the VBR reservation model where processes reserve an average rate and a maximum burst size. In this model, message sending is divided into slots of a fixed size, Θ, which is the same for all processes and is fixed throughout the execution of the algorithm. In addition, there exists a well-known constant k which is the number of slots over which the average is computed. Specifically, the application declares two rate parameters:

1. $NetAvgRate$ – the average message rate per Θ time. This means that $k * NetAvgRate$ is the maximum number of messages that may be sent during $k * \Theta$ time.

2. $NetMaxBurst$ – the maximum number of messages that may be sent during Θ time.

The reservation request m has two fields: $NetAvgRate$ and $NetMaxBurst$. The reservation request interface for this model is $reserve(m, \Theta, \Delta)_i$. Since all the processes reserve the same Θ and the same Δ, we omit these parameters and simply write $reserve(m)_i$. Note that although Θ is fixed and known to all processes in each application, different applications can use different slot sizes.

In both reservation models, if the application violates its declared rate then there is no delay guarantee from the network for these messages, and, furthermore, some messages may be dropped. Throughout this paper we assume that the application does not violate its declared rates; thus the guarantees still hold and we can analyze the delay of the TO service as a function of the network delay.

In Sections 4 and 5 we assume that no process failures or joins occur during the execution of the algorithm, that is, a fixed set of processes run the algorithm. Furthermore, we assume that the network is reliable (i.e., there are no message losses). We discuss process failures and joins in Section 6 and message loss in Section 7.

2.2 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. Throughout this paper we denote by *now* the real time that has passed from the beginning of the execution[2], thus, each execution starts with $now = 0$. We assume that the maximum difference between $clock_i$ and *now* is at most $\Gamma/2$, thus, we have for each process i:

$$now - \Gamma/2 \leq clock_i \leq now + \Gamma/2$$

This inequality is assumed to hold throughout the execution of the algorithm. Note that this implies that the maximum difference between two processes' internal clocks is at most Γ. In this paper we represent the algorithm as timed automata (cf. [27], Ch. 23), and assume that the processes' clocks guarantee the above properties.

The assumption that clocks are synchronized within a bound is very reasonable. For example, the *Network Time Protocol (NTP)* [32] 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. For this assumption to hold, operating system support for real-time scheduling is required (for examples, see [8, 9, 25, 20, 18, 40, 24]).

3 Totally Ordered Multicast Service Specifications

The algorithms we present in this work guarantee total ordering of messages and preserve QoS parameters.

3.1 Total Order

We present two different semantics of totally ordered multicast.

In the following definitions, we say that a message is sent by a process i when $app\text{-}mcast(i)$ occurs; that i delivers a message m when the total order service at process i performs the $deliver(m)_{i,k}$ action, for a message m sent by the application at process k. We say that i receives a message m when the network performs the $net\text{-}receive(str, m)_{i,k}$ action for a message m sent by the application at process k. In absence of recoveries, processes that do not crash are considered correct. In Section 6 we explain when recovering processes are considered to be joining the algorithm, that is, when they become "correct" for the sake of the definitions below.

We first define reliable (gap-free) total order (sometimes called Atomic Broadcast [16]):

Definition 1 *Reliable Total Order*

- 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 more times than it was sent.*
- Liveness: *Every message sent or delivered by some correct process is eventually delivered by every correct process.*

[2] The real time is used as an abstraction for the analysis – we analyze the latency guarantees in terms of the real time that has elapsed.

This definition implies that throughout the execution of the algorithm, for every two processes i and j, if j delivers at least as many messages as i, then the sequence of messages delivered by i is a prefix of the sequence of messages delivered by j.

The following definition weakens the liveness condition of the previous one, to allow for gaps in the total order. The safety conditions (Total Order and Integrity) remain the same.

Definition 2 *Total Order with Gaps:*

- Total Order *and* Integrity: *As in Definition 1.*
- Liveness: *Every message that is sent by some correct process j and received by a correct process i, is eventually delivered by i.*

Note that the liveness condition here depends on the liveness of the underlying network, since the total order algorithm is required to deliver messages from correct process that were received from the underlying network. Thus, although gaps in the total order are permitted, the dropped messages must have been sent by faulty processes or omitted by the underlying network.

Total order with gaps is provided by several group communication systems (e.g., Ensemble [17], Horus [13], Phoenix [28], RMP [42], Totem [3, 33] and Transis [7]). The algorithms of [2, 12, 14, 21, 22] are implemented atop group communication systems providing total order with gaps, and extend them to eventually provide reliable total order under certain conditions on failures in the underlying network. However, these algorithms do not provide latency guarantees.

3.2 QoS parameters

The sending application declares its transmission rate using the $\textit{init-rate}(m)_i$ action at the beginning of the algorithm. The application can renegotiate its transmission rate using the $\textit{new-rate}(m)_i$ action, at any time during the execution. We denote the rate requested by the application in the CBR model by *AppRate*, and the rates of the VBR model by *AppAvgRate* and $AppMaxBurst$. These are dual to the respective network QoS parameters.

Our algorithms guarantee a bound on the maximum latency on message delivery to the application: *AppLatency*. *AppLatency* is the supremum over all executions, all messages m and all processes i from the time $\textit{app-mcast}(m)_i$ action is performed by process i in some execution until m is ready to be delivered by all processes that deliver it. That is, the time at which the action $\textit{deliver}(m)_{j,i}$ is enabled for every process j that performs the $\textit{deliver}(m)_{j,i}$ action. We ignore the local scheduling time until *deliver* is executed, for simplicity. Throughout the paper, we analyze *AppLatency* as a function of Δ. Assuming the network guarantees an upper bound Δ on message latency, we analyze the upper bound, *AppLatency*, that the total order algorithm can guarantee on message delivery time.

4 CBR

In the CBR model the application declares a rate of m messages per second, so that it will send at most one message each $1/m$ second. We assume no process failures or joins and no message loss. We present a totally ordered multicast algorithm, and then analyze its QoS guarantees and show its correctness.

4.1 The algorithm

The algorithm we present is symmetric – all the processes implement the same algorithm to decide which message to deliver next to the application. Since we assume that no process fails, we could actually have each process deliver all messages from other processes in round robin order, for example deliver a message from 1, then from 2 etc. However, since we assume that processes have different transmission rates, this would cause messages sent by processes with high transmission rates to be delayed by processes with low transmission rates. For example consider the case that process 1 sends 10 messages per second and process 2 sends 100 messages per second. In this case, messages from process 2 would take longer and longer to get delivered as the execution progresses. Thus, the algorithm introduces unbounded delays, which violate QoS guarantees and also require unbounded buffer space.

To remedy this shortcoming, Chockler et al. [7] have suggested an algorithm that explicitly uses the process sending rates to agree upon the order in which messages are delivered. Assuming that the relative sending rates are as above and are known to the processes, those rates can be exploited as follows: all the processes deliver one message from process 1 for every ten messages they deliver from process 2. The algorithm in [7] tracks process sending rates to predict future rates. In our model, this is not necessary since we assume that processes declare their rates initially, and whenever they change their rates. Our algorithm is therefore a variation on the algorithm of [7], tailored to preserve the QoS parameters of our model.

We present our algorithm as a collection of identical timed automata, one for each process i. The automaton may be in several modes, tracked by the state variable *mode*. The mode changes of the automaton are depicted in Figure 2. We present the timed automaton implementing this algorithm in Figures 3 and 4. In this algorithm, M stands for the set of possible messages and N denotes the natural numbers. The automaton for this algorithm works in the following way: First, the application must declare its initial transmission rate using the $\mathit{init\text{-}rate}(m)_i$ action. The automaton then moves to the $\mathit{FirstReserve}$ mode where $\mathit{reserve}(m)$ occurs, and then, the mode changes to $\mathit{DetOrder}$. Process i then multicasts its rate to all processes, and waits to receive the initial rate messages from all the other processes. When i receives the transmission rates from all other processes, it computes a vector, Porder, that determines the order in which messages will be delivered. Porder is computed in the Update function as we explain below. The process then executes the internal action $\mathit{start\text{-}deliver}$, and changes its mode to $\mathit{DeliverMessages}$.

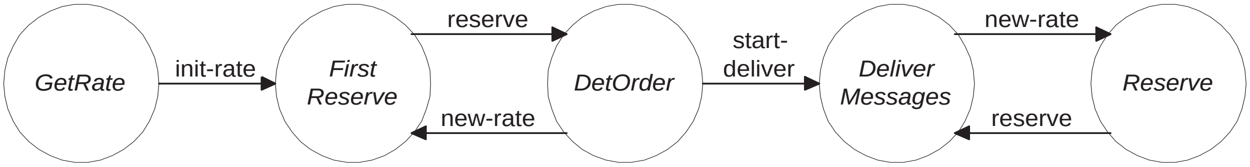


Figure 2: Mode changes in the CBR algorithm.

If the application at process i declares a transmission rate of m messages per second, then the total order algorithm at process i sends a message each $1/m$ second. When the application does not send at the reserved rate, the algorithm uses explicit time-outs to send dummy messages (using the *dummy* internal action). When messages are received, they are stored in a local buffer, *Rqueue*, per source. Messages are delivered from *Rqueue* to the application using the $\mathit{deliver}(m)_{i,j}$ action according to the Porder vector. That is, the algorithm first delivers a message from the *Rqueue* buffer of process number $\mathit{Porder}[0]$, then from $\mathit{Porder}[1]$, etc. After the last entry of Porder, it cycles back to $\mathit{Porder}[0]$. If the next message ready for delivery is a dummy message, it is discarded (this is done in the *handle-dummy* action).

When the application performs the $new\text{-}rate(m)_i$ action, i changes mode to *Reserve* (or, if in *DetOrder*, *FirstReserve*), where it reserves the new rate using the $reserve(m)_i$ action before returning. Process i then multicasts the new rate as its next message. If the next message ready for delivery according to *Porder* is a new rate message from another process, the algorithm recomputes its *Porder* vector in the *change-rate* action. From this point on, it delivers messages according to the new vector it computed.

Signature

Input:
- $app\text{-}mcast(m)_i$, $m \in M$
- $init\text{-}rate(m)_i$, $m \in N$
- $new\text{-}rate(m)_i, m \in N$
- $net\text{-}receive(str, m)_{i,j}$, $str \in$ String, $m \in (N \cup M)$

Output:
- $deliver(m)_{i,j}$, $m \in M$
- $net\text{-}mcast(str, m)_i$, $str \in$ String, $m \in (N \cup M)$
- $reserve(m)_i$, $m \in N$

Time-passing:
- $v(t)$, $t \in R^+$

Internal:
- $update\text{-}rate_j$
- *dummy*
- *handle-dummy*
- *start-deliver*
- *change-rate*

State

Squeue, a FIFO queue of messages, initially empty
For all j, *Rqueue*(j), a FIFO queue of messages, initially empty
rate, an array of integers (one for each process), initially $\perp$ in all places
myRate, an integer, initially $\perp$
Porder, vector of process indices, initially of length 0
current, an integer // The current place in *Porder*
$clock \in R^{\geq 0}$, initially 0
$last \in R^+ \cup \{\infty\}$, initially ∞
$mode \in \{GetRate, FirstReserve, DetOrder, DeliverMessages, Reserve\}$, initially *GetRate*
derived *total* = sizeof(Porder), an integer // The size of *Porder*

Figure 3: The CBR algorithm automaton for process i: signature and variables.

The automaton uses the *Update* internal function presented in Figure 5 to compute the order of the messages. The parameters P and *cur* are passed to *Update* by reference, and *Update* changes them. *Update* is used at startup – to create the initial order vector – and it is subsequently used to create a new order when a process changes its rate. At startup, *Porder* is first set to have rate[1] entries, all containing 1. Then *Update* is called with n set to 2 to insert the rate of process 2, and so on until process n is inserted.

The *Update* function constructs the *Porder* vector, which determines the delivery order of messages sent during one second. Consider the set of pairs (i, k) where $1 \leq i \leq n$ is a process index and $0 \leq k \leq (rate[i] - 1)$ is i's $k + 1$st message sent in a second. This set corresponds to the set of messages sent by all processes in one second. *Porder* defines an order on these pairs so that if $k/rate[i] < k'/rate[i']$ then (i, k) precedes (i', k') in *Porder*.

If the clocks of i and i' are perfectly synchronized, then $k/rate[i] < k'/rate[i']$ implies that k was sent before k'. Thus, if every process sends according to the rate it specified, and processes' clocks are perfectly synchronized, in order to deliver a particular message m, a process needs to wait only for messages that were sent at the same time as m or earlier. Since we assume that clocks are synchronized within Γ, this implies that, in order to deliver a particular message m, a process needs not wait for messages that were sent later than Γ time after the sending time of m.

Transitions

```
**Input** $init\text{-}rate(m)_i$
    Eff: if ($myRate = \perp$) {
            add ("$init\text{-}rate$", $m$) to $Squeue$
            $myRate := m$
            $mode := FirstReserve$
            $last := clock + 1/myRate$
         }

**Output** $reserve(m)_i$
    Pre: $mode = FirstReserve$ or $mode = Reserve$
         $m = myRate$
    Eff: if ($mode = FirstReserve$)
            $mode := DetOrder$
         else
            $mode := DeliverMessages$

**Internal** $update\text{-}rate_j$
    Pre: ("$init\text{-}rate$", $m$) is first on $Rqueue(j)$
    Eff: $rate[j] := m$
         discard first element of $Rqueue(j)$

**Internal** $start\text{-}deliver$
    Pre: $mode = DetOrder$
         for all $1 \leq j \leq n$ $rate[j] \neq \perp$
    Eff: $Porder[1...rate[1]] := 1$
         for each $j$ in $(2...n)$
           $Update(Porder, 0, rate, j, j)$
         $current := 0$
         $mode := DeliverMessages$

**Input** $app\text{-}mcast(m)_i$
    Eff: if ($mode \neq GetRate$)
           add ("$message$", $m$) to $Squeue$

**Output** $net\text{-}mcast(str, m)_i$
    Pre: $(str, m)$ is first on Squeue
    Eff: $last := last + 1/myRate$
         remove first element of $Squeue$

**Internal** $dummy$
    Pre: $clock = last$
    Eff: add("$dummy$", "$dummy$") to $Squeue$
```

```
**Input** $net\text{-}receive(str, m)_{i,j}$
    Eff: add $(str, m)$ to $Rqueue(j)$

**Output** $deliver(m)_{i,j}$
    Pre: $mode = DeliverMessages$
         $j = Porder[current]$
         ("$message$", $m$) is first on $Rqueue(j)$
    Eff: $current := (current + 1)$ mod $total$
         discard first element of $Rqueue(j)$

**Internal** $handle\text{-}dummy$
    Pre: $mode = DeliverMessages$
         $j = Porder[current]$
         ("$dummy$", $m$) is first on $Rqueue(j)$
    Eff: $current := (current + 1)$ mod $total$
         discard first element of $Rqueue(j)$

**Input** $new\text{-}rate(m)_i$
    Eff: if ($mode \neq GetRate$) {
           add ("$new\text{-}rate$", $m$) to $Squeue$
           $myRate := m$
           if ($mode = DetOrder$)
               $mode := FirstReserve$
           else
               $mode := Reserve$
         }

**Internal** $change\text{-}rate$
    Pre: $mode = DeliverMessages$
         $j = Porder[current]$
         ("$new\text{-}rate$", $m$) is first on $Rqueue(j)$
    Eff: $rate[j] := m$
         $Update(Porder, current, rate, j, n)$
         discard first element of $Rqueue(j)$

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

Figure 4: The CBR algorithm automaton for process i: transitions.

When the new *Porder* vector is computed, *current* is also updated according to the new rate. We would like to continue to deliver messages from the place we were before we updated *Porder*. Thus, when recomputing *Porder* we update *current* to point to the next message that we should have delivered had we not updated *Porder*.

```
algorithm for adding the new rate of process $j$
Update(vector $P$, integer $cur$, array $rate$, integer $j$, integer $n$) {
    $deliveredJ = \#$ (appearances of $j$ in $P$ until $cur$)
    $PrateJ = \#$ (appearances of $j$ in $P$)                // $j$'s previous rate
    $cur := cur - deliveredJ$
    $P := P$ without all the appearances of $j$
    $total := \sum_{i=1}^{n} rate[i]$
    $temp :=$ new vector of size $total$
    $count :=$ new array of size $n$, initially 0 in all places
    $oldIndex = 0$
    for($newIndex := 0$ to $total - 1$) {
        $i = P[oldIndex]$
        if ($count[j]/rate[j] < count[i]/rate[i]$) {
            $temp[newIndex] := j$
            $count[j]++$
            if ($count[j]/rate[j] \leq deliveredJ/PrateJ$) $cur++$
        } else {
            $temp[newIndex] := i$
            $count[i]++$
            $oldIndex++$
        }
    }
    $P := temp$
}
```

Figure 5: The CBR algorithm automaton: internal function *Update*.

4.2 Using receiver based reservation

The above algorithm uses a sender based reservation protocol. It is easy to modify the algorithm to use a receiver based reservation protocol. Since in our algorithm the sender sends its rate parameters each time they change, we can simply have the receiver reserve the new rate when it receives this rate. Since the sender cannot know exactly when the reservation occurs, it must send at the minimum of the old rate and the new one until it learns that the reservation was successful. If the new rate is slower than the old rate, the receiver immediately changes its *Porder* vector accordingly. Otherwise, the receiver sends a message to inform the sender that the reservation was successful. Once the sender receives such messages from all the processes, it sends a special message to denote that it is now starting to use the new rate, and following it, sends at the new rate. When such a new rate message is received, the receiver changes its *Porder* vector.

This modification does not affect the latency of the algorithm. In both cases, the sender's rate changes exactly after it sends the message that causes *Porder* to change at the receivers. The time from when a process sends its new rate message until it actually starts using this rate is part of the renegotiation time, and is not taken into account in our delay analysis.

4.3 Correctness of the algorithm

In order to prove the correctness of this algorithm we show that Definition 1 holds. Integrity is trivially satisfied from our assumptions on the network. Liveness is satisfied since all the processes continuously send messages and eventually all messages arrive at each process. Since *Porder* contains entries for all the processes, all the messages eventually get delivered. The following two lemmas prove that Total Order is also preserved by this algorithm.

We say that process i *processes* a message m when i executes an action that removes the message from *Rqueue*. In our algorithm, a message (other than an *init-rate* message) can be processed by one of the following three message processing actions: $deliver(m)_{i,j}$, *handle-dummy* and *change-rate*. In the following lemmas we use the following notations: Denote by $Porder_i^k$ $(k \geq 1)$ i's *Porder* vector immediately before i processes its kth message (that is, the vector according to which i's kth message is processed). Denote by $current_i^k$ the value of i's *current* variable immediately before i processes its kth message.

Lemma 4.1 *For any process j, $Porder_i^k = Porder_j^k$ (where this equality means that the vectors are identical) and $current_i^k = current_j^k$.*

Proof: By induction on k. Base $(k = 1)$: i processes its first message after it computes *Porder* for the first time, because invoking each of these functions requires $mode = DeliverMessages$ which does not happen until i calculates *Porder*. When i processes its first message, its order is based on the first rate message sent to it by each process. This is so because i updates its *rate* array before it computes *Porder* only upon receipt of an *init-rate* message from another process. Since such a message is sent only once by each process, when computing *Porder* for the first time (in i's *start-deliver* internal action), i's order is based on the first rate message sent by each process. Since the channels ensure FIFO ordering of messages from each process, j will also see the same rates for all the processes (the *init-rate* messages are the first message sent by each process) and will have the same vector *Porder* as i's initial vector (they both calculate this order using the same deterministic function, and so will have the same vector as a result). Since *current* is set to 0 after computing the order for the first time, both processes will have the same value for *current* when they process their first message.

Inductive step: Assume the kth message i processes is from p, and assume this is the nth message sent by p that i received. Since p multicasts its messages, and the channels preserve FIFO order, this will also be the nth message from p that process j receives. But if i processes $n-1$ messages from p prior to processing this message, then it must be that j processes exactly the same number of messages from p before it processes its kth message, because each time i processes a message from p prior to the kth message it processes, j also processes a message from p, since according to the induction hypothesis, their vector and place in it are the same for each previous processing of a received message. So when j processes its kth message, it also sees the nth message from p. Now, if the nth message from p is a regular message of the application or a *dummy* message, then both processes will advance *current* by 1 and will not change their *Porder* vector, and so $Porder_i^{k+1} = Porder_j^{k+1}$ and $current_i^{k+1} = current_j^{k+1}$. If the nth message from p is a *new-rate* message, then both j and i will use the function *Update* to calculate a new vector based on p's new rate. But since they both had the same vector previously, and both have the same new rate for p (and apart from that all the rates stay as before), they will both have the same *Porder* vector and *current* value after executing the *Update* function. So in both cases i and j will have the same *Porder* vector and current value after processing their kth message. ■

Lemma 4.2 *The kth message delivered by i is the kth message delivered by j.*

Proof: Denote by r the number of messages i processed before delivering its kth message (the kth message it delivers is the $r+1$ message it processed). Then, according to the previous lemma this is also the $r+1$ message that j processes. For each message i processed until this message, j processed the same message and so for every message i delivered, j delivered the same message. So after processing r messages j also delivered $k-1$ message, and so the $r+1$ message is also the kth message j delivers. ■

The Total Order of Definition 1 follows directly from Lemma 4.2: since for each k, the kth message delivered by i is the kth message delivered by j, then if i and j both deliver the same two messages m and m', they deliver these messages in the same order.

4.4 QoS guarantees of the algorithm

We now analyze the QoS guarantees of the algorithm. We first analyze the latency, and then the transmission rate.

Theorem 1

$$AppLatency = \Delta + \Gamma$$

Proof: We constructed our ordering of message delivery based on the transmission rate of each process. Our algorithm ensures that each process actually sends according to its rate (using *dummy* messages if the application has nothing to send). Thus, the ordering in $Porder$ guarantees that in order to deliver a message m we have to wait only for messages that were sent before m or at the same time according to the senders' local clocks. The possible difference between process clocks may add at most Γ to the delay. So in order to deliver m we have to wait for messages that were sent at most Γ time after m. According to the network guarantees, every such message will arrive at every process in at most Δ time. So in order to deliver m, a process has to wait at most $\Delta + \Gamma$ from the time the message was sent to receive all message that are ordered before it. Note that this latency can actually occur, so this is the maximum possible latency. ■

Rate: If the application specifies a certain message rate *AppRate*, our algorithm uses the same rate as specified by the application, and so we have:

$$NetRate = AppRate$$

Our algorithm does not induce any overhead in addition to the reserved rate, with the exception of in-band rate messages sent at start-up time and at renegotiation time. Since renegotiation is expected to be rare, and is known to be costly for reasons external to our algorithm, we ignore the extra overhead induced by such messages.

Note that although the total order protocol of this section does not increase the reserved rate, it does increase the load by sending dummy messages in case the application sends at a lower rate than it reserved. In general, the CBR model is costly since typical applications do not transmit at a uniform rate. This shortcoming is addressed by the VBR model discussed in the next section.

5 VBR

In the VBR model, there is a fixed slot size, Θ, which is known to all of the processes. The application declares two parameters, $AppAvgRate$ and $AppMaxBurst$ per Θ time. In this section we assume no process failures or joins and no message loss.

5.1 The VBR algorithm

We present a symmetric algorithm, in which all the processes deliver messages according to the same rule. The algorithm automaton is presented in Figures 6 and 7. In this algorithm we use slots for sending and delivering messages.

Signature

Input:
- *app-mcast*$(m)_i$, $m \in M$
- *init-rate*$(avg, max)_i$, $avg, max \in N$
- *new-rate*$(avg, max)_i$, $m \in N$
- *net-receive*$(str, m)_{i,j}$, $str \in$ String, $m \in (N \cup M)$

Output:
- *deliver*$(m)_{i,j}$, $m \in M$
- *net-mcast*$(str, m)_i$, $str \in$ String, $m \in (N \cup M)$
- *reserve*$(m)_i$, m a two field variable

Time-passing:
- $v(t), t \in R^+$

Internal:
- *update-rate*$_j$
- *dummy*
- *handle-dummy*
- *start-deliver*
- *change-rate*
- *next-slot*

State

Squeue, a FIFO queue of messages, initially empty
For all j, *Rqueue*(j), a FIFO queue of messages, initially empty
current, an integer initially 1, // current process to receive from
count, an integer initially 0, // number of messages received from *current* in this slot
myCount an integer initially 0, // number of messages sent in this slot
slot, an integer initially 1, // current slot number
maxRate, array of size n of integers, initially $\perp$ in all places
myAvg, *myMax*, integers initially $\perp$
maxSend, integer initially $\perp$ // i's *AppMaxBurst* rate
$clock \in R^{\geq 0}$, initially 0
$last \in R^+ \cup \{\infty\}$, initially ∞
$mode \in \{GetRate, DetRate, DeliverMessages\}$, initially *GetRate*
changeRate, boolean, initially FALSE // indicates that *reserve* has to happen

Figure 6: The VBR algorithm automaton for process i: signature and variables.

As in the CBR algorithm, the application has to perform the *init-rate* action, this time stating two parameters: *AppAvgRate* and *AppMaxBurst*. The algorithm reserves the following rates from the network: $NetMaxBurst = AppMaxBurst$, and $NetAvgRate = AppAvgRate + 1/\Theta$. The $reserve(m)_i$ action also places a ("*rate*", *AppMaxBurst*) message at the head of the sending queue, and thus, *AppMaxBurst* is sent to all the processes. While both *AppAvgRate* and *AppMaxBurst* are used for specifying the reserved QoS, only *AppMaxBurst* effects the message ordering, and therefore, only *AppMaxBurst* is sent to other processes. The *AppMaxBurst* values are stored in the array *maxRate* by the *update-rate*$_j$ action.

Processes use slots for sending and delivering messages. The algorithm sends (using *net-mcast*) application messages to all processes in the same slot as they were *app-mcast* by the application. When a slot ends (that is, Θ time has passed from the time it started) the process checks the number of messages it sent during this slot. If this number is equal to its *AppMaxBurst*, then it moves to the next slot using its *next-slot* action. Otherwise, it sends a *dummy* message as the last message in the slot, (in the *dummy* action) and then moves to the next slot.

Transitions

```
Input $init\text{-}rate(avg, max)_i$
    Eff: if ($myAvg = \perp$) {
            $myAvg := avg + 1/\Theta$
            $myMax := max$
            $last := clock + \Theta$
            $changeRate = TRUE$
            $mode := DetRate$
         }

Output $reserve(m)_i$
    Pre: $myCount = 0$
         $changeRate = TRUE$
         $m = (myAvg, myMax)$
    Eff: add (“$rate$”, $myMax$) as first element in $Squeue$
         $maxSend := myMax$
         $changeRate := FALSE$
         $myCount{+}{+}$

Internal $update\text{-}rate_j$
    Pre: (“$rate$”, $max$) is first on $Rqueue(j)$
         $maxRate[j] = \perp$
    Eff: $maxRate[j] := max$
         discard first element of $Rqueue(j)$

Internal $start\text{-}deliver$
    Pre: $mode = DetRate$
         for all $1 \le j \le n$
             $maxRate[j] \ne \perp$
    Eff: $mode := DeliverMessages$

Input $app\text{-}mcast(m)_i$
    Eff: if ($mode \ne GetRate$)
             add (“$message$”, $m$) to $Squeue$

Output $net\text{-}mcast(str, m)_i$
    Pre: $(str, m)$ is first on $Squeue$
         $myCount < maxSend$
         $changeRate = FALSE \lor myCount > 0$
    Eff: if ($str = dummy$)
             $myCount := maxSend$
         else
             $myCount{+}{+}$
         remove first element of $Squeue$

Internal $dummy$
    Pre: $clock = last$
         $myCount < maxSend$
         $Squeue$ is empty
    Eff: add (“$dummy$”, “$dummy$”) to $Squeue$
```

```
Internal $next\text{-}slot$
    Pre: $clock = last$
         $myCount = maxSend$
    Eff: $last := clock + \Theta$
         $slot{+}{+}$
         $myCount := 0$

Input $net\text{-}receive(str, m)_{i,j}$
    Eff: add $(str, m)$ to $Rqueue(j)$

Output $deliver(m)_{i,j}$
    Pre: $mode = DeliverMessages$
         $j = current$
         (“$message$”, $m$) is first on $Rqueue(j)$
    Eff: $count{+}{+}$;
         if ($count = maxRate[j]$) {
             $count := 0$
             $current := (current + 1) \bmod n$
         }
         discard first element of $Rqueue(j)$

Internal $handle\text{-}dummy$
    Pre: $mode = DeliverMessages$
         $j = current$
         (“$dummy$”, $m$) is first on $Rqueue(j)$
    Eff: $current := (current + 1) \bmod n$
         $count := 0$
         discard first element of $Rqueue(j)$

Input $new\text{-}rate(avg, max)_i$
    Eff: if ($mode \ne GetRate$) {
             $myAvg := avg + 1/\Theta$
             $myMax := max$
             $changeRate := TRUE$
         }

Internal $change\text{-}rate$
    Pre: $mode = DeliverMessages$
         $j = current$
         (“$rate$”, $max$) is first on $Rqueue(j)$
    Eff: $maxRate[j] := max$
         $count{+}{+}$

TimePassage $v(t)$
choose $p \ge 0$
    Pre: $now + t - \Gamma/2 \le clock + p \le now + t + \Gamma/2$
         $clock + p \le last$
    Eff: $now := now + t$
         $clock := clock + p$
```

Figure 7: The VBR algorithm automaton for process i: transition definitions.

For each slot, the algorithm delivers messages according to the process indices: it delivers all the messages for this slot sent by process 1, then all the messages sent by 2 etc. This is done in the $deliver(m)_{i,j}$ action. If process i is currently delivering messages for slot s from process j, it will move to deliver messages from $(j+1 \bmod n)$ if it sees a *dummy* message from j (in its *handle-dummy* internal action), or upon delivery of $maxRate[j]$ messages from j in this slot.

The application can perform the *new-rate* action at any time to change the rate parameters. However, the new rates will be used only from the ensuing slot. Each time the application changes its rate, the algorithm uses its $reserve(m)$ action to reserve the new rates. The $reserve(m)$ action also places a (*"rate"*, *AppMaxBurst*) message at the head of the sending queue. This message is sent as the first message in the next slot. When a rate message is the next message to be delivered, the receiving process starts delivering messages according to the new rate and discards this message (using its *change-rate* action).

5.2 Correctness

In order to prove the correctness of this algorithm we have to show that Definition 1 holds. Integrity is trivially satisfied from our assumption that the network does not duplicate, corrupt or spontaneously generate messages. The following lemmas show that Total Order and Liveness hold.

Lemma 5.1 *Denote by $maxSend_i^s$ the value of i's maxSend variable when i terminates slot s (using the next-slot action). Denote by $maxRate[i]_j^s$ the value of $maxRate[i]$ at j when j delivers the last message i sent in slot s. Then, $maxSend_i^s = maxRate[i]_j^s$.*

Proof: By induction on s. Base ($s=1$): i's initial $maxSend$ value is the value i declared in its *init-rate* action. Denote this value by r. ("rate",r) is sent as i's first message in slot 1, and all processes wait to receive this value before they start delivering messages. So r is the first value j has for $maxRate[i]$. After adding this message to i's *Squeue*, i increments *myCount* (in the $reserve(m)_i$ action). Since the precondition for sending a rate message and changing i's *maxSend* value is that $myCount = 0$, i will not send any other rate message in slot 1 (because *myCount* is set to 0 again only after i finishes slot 1). So $maxSend_i^1 = r$. Since j changes $maxRate[i]$ only when it processes a rate message from i, j will not change its initial value of $maxRate[i]$ (which is r) when it delivers i's messages for slot 1. So $maxRate[i]_j^1 = r$.
Inductive step: If the first message i sent in slot $s+1$ is not a rate message, then as explained above i will not send any rate message during slot $s+1$, and will not change its *maxSend* value in this slot. Since j only changes its $maxRate[i]$ value after processing a rate message from i, j will not change its $maxRate[i]$ value when delivering i's messages for slot $s+1$. Combining this with the induction hypothesis we get $maxSend_i^{s+1} = maxRate[i]_j^{s+1}$.
If the first message i sent in slot $s+1$ is a ("rate",r) message, then as explained above r will be i's *maxSend* value at the end of this slot, and so $maxSend_i^{s+1} = r$. Since j will process this message as the first message i sent for this slot, j will set its maxRate[i] value to r after processing i's first message for slot $s+1$. This is the only rate message j will receive from i for this slot, and so $maxRate[i]_j^{s+1} = r$. ■

Lemma 5.2 *Let s be a slot. Assume that each process delivers all the messages sent before slot s before delivering any message sent in slot s. Then, all the processes deliver all the messages sent in slot s in the same order, before delivering any messages sent in later slots.*

Proof: When process j delivers process i's messages for slot s, j waits to receive all of i's messages for this slot before it moves to deliver messages from the next process. This is guaranteed by the fact that j will not move on before it received a *dummy* message from i or it already delivered $maxRate[i]$ messages from i, and from Lemma 5.1, these are exactly all the messages i sent in s.

This is true for all processes, and since the network preserves the FIFO order and messages are not lost, all processes will deliver the same messages from i in for slot s, and in the same order. Since the processes deliver messages in each slot according to a predefined process order (first from 1 then from 2 and so on), all processes will deliver messages in the same order for slot s. ■

This lemma is true for each slot s, and so we can show by simple induction on s (the slot number) that if i and j both delivered s slots, they delivered the same set of messages, and in the same order.

5.3 QoS guarantees

The maximum delay caused by this algorithm is the following:

Theorem 2

$$AppLatency = \Delta + \Gamma + \Theta$$

Proof: We only add a *dummy* message (to end a slot) after the time for this slot is over, and *Squeue* is empty. So all messages that were *app-mcast* before the *dummy* message are *net-mcast* before it. If the application sends *AppMaxBurst* messages for slot s, we do not add a *dummy* message and all messages are *net-mcast* before the slot ends. So in both cases (using a *dummy* message to end a slot or *AppMaxBurst* messages are sent by the application), if a message is *app-mcast* before the time at which slot s ends, then it will be *net-mcast* in slot s. So from now on when we say that a message was 'sent' in slot s it means that both *app-mcast* and *net-mcast* were performed in that slot.

Assume that process i sends a message m in slot s, and the delivery of m is delayed until a message m' from another process, j, will be received and delivered. Since message delivery is done per slot (see Lemma 5.2 above), m' must be sent by j before the end of slot s. Since the difference between the two processes' internal clock is at most Γ, we know that i sent m (in slot s) at most $\Gamma + \Theta$ time before j sent its last message for s (see Figure 8). Since j's last message arrives at all the processes at most Δ time after it is sent, all the processes receive m' at most Δ time after j's last message for slot s was sent. Thus, after at most $\Delta + \Gamma + \Theta$ time from the time i sent m, its delivery will be enabled at all the processes. Since the scenario illustrated in Figure 8 can actually happen, $\Delta + \Gamma + \Theta$ is the maximum delay for any message sent in this algorithm.

■

The algorithm reserves rates that are a function of the application specified rates. The following theorem proves that the rates reserved by the algorithm guarantee that the application messages will have the required bandwidth. That is, enough bandwidth is reserved to allow both the sending of the application messages and the extra messages that are added by the algorithm.

Theorem 3

$$NetAvgRate = AppAvgRate + 1/\Theta$$

$$NetMaxBurst = AppMaxBurst$$

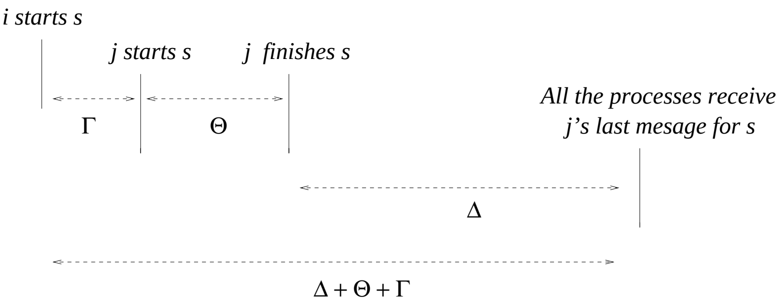


Figure 8: Maximum delay with the VBR algorithm.

Proof: The only message this algorithm adds to the messages sent by the application is the *dummy* message. This message is sent at most once every slot as the last message in the slot. Thus, the message rate is increased by at most one message each slot, and so the average rate is increased by $1/\Theta$. If the application sent $AppMaxBurst$ messages in slot s, the algorithm does not add any message of its own, and so $AppMaxBurst$ remains the same as the one declared by the application. ■

Note the interesting tradeoff between the increase in the delay and the increase in the transmission rate. A bigger Θ will cause increased delay while decreasing the rate, while a smaller Θ gives rise to a smaller delay, but increases the transmission rate.

6 Process Failures and Joins

In this section we extend the VBR model of the previous section to allow for process failures and joins.

6.1 A lower bound on reliable total order with process failures

The following theorem shows that in this model any algorithm implementing reliable total order (Definition 1 of TO) can guarantee, at best, a latency bound which is proportional to the number of failures it can tolerate.

Theorem 4 *A reliable total order algorithm that can tolerate f process failures cannot guarantee a latency bound smaller than $(f+1)\Delta$.*

Proof: Assume that a reliable total order algorithm $\mathcal{A}$ can tolerate f process failures and guarantees a latency bound of δ. We now show that $\delta \geq (f+1)\Delta$. As shown in [16], the processes may use $\mathcal{A}$ to solve the Consensus problem by sending their initial values as their first messages and agreeing upon the value in the first delivered message. By our assumption on $\mathcal{A}$, this message is delivered at most δ time after the algorithm is initiated, and thus, Consensus is solved in δ time.

Since $f+1$ rounds is a well known lower bound for synchronous Consensus tolerating f stopping failures (see [27], Ch. 6.7), and from our assumption that messages can be delayed up to Δ time in the network, we conclude that the algorithm cannot guarantee that Consensus be solved in less than $(f+1)\Delta$ time, and hence $\delta \geq (f+1)\Delta$. ■

6.2 Total order with gaps: Detecting faults

The above theorem proves that the latency achieved by a reliable total order algorithm is bound to be proportional to the number of faults tolerated. In order to achieve delays that are constant no matter how many processes fail, we relax the reliable total order definition and consider total order with gaps (Definition 2) for the rest of this section and the next one. Since in this section we do not consider message loss, gaps in the total order can only correspond to messages of faulty processes. For example, if process i sends a message m and immediately fails, then it is possible that some correct process will deliver m and another will not.

Since we allow for such gaps, the algorithm can deliver messages without checking if other processes received these messages too. Processes deliver all the messages that they do receive according to the order specified in the previous section. For example, we do not want a situation in which after i starts delivering messages from process $j+1$ for slot s it receives a new message from j for slot s. In order to overcome faults and yet avoid this situation we implement an internal failure detector at each process as explained below.

According to the network guarantees, if a process sends a message all other processes will receive it in at most Δ time. This allows us to implement a failure detector at each process in the following way: We say that i finished slot s when i executes its internal *next-slot* action and increments its slot number to $s+1$. If a process i waits more than $\Delta + \Gamma$ time from the time it finished slot s, for a message from another process for this slot, then it knows that this process 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 will be described later), i does not deliver messages from j. This requires i to keep track of the slot it is delivering messages for.

6.3 Allowing new processes to join

For a joining process we allow an *initialization time*. During this time the process does not have to deliver all messages it receives (it is not considered "correct" for Definition 2). After the initialization time, the process joins the algorithm, and is required to deliver every message it receives as per Definition 2. We now analyze the initialization time as a function of the Δ, Θ and Γ.

When a process j wants to join the algorithm, it just needs to know in which slot the rest of the processes are in order to start sending messages. To do so, 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 $(\text{"}join\text{"}, j, r, s)$ message to all processes where r is j's rate and 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 with the rate r. 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 processes will receive it before they start slot $Smax = (clock_j + \Delta + \Gamma)/\Theta + 1$. Thus, j chooses $Smax = (clock_j + \Delta + \Gamma)/\Theta + 1$ as the slot it will join in, sends $(\text{"}join\text{"}, j, r, Smax)$ to all processes, and starts sending and delivering messages from slot $Smax$.

When a process i receives j's $(\text{"}join\text{"}, j, r, Smax)$ message, it adds j to its list of active processes upon starting to deliver messages for slot $Smax$. Then, with the first message in slot $Smax$, i includes the slot number ($Smax$) and its own transmission rate. Thus, j learns where the other processes start $Smax$ and finds out their rates before starting to deliver their messages. 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. Thus, every other process starts this slot at most $\Delta + 2\Gamma + \Theta$ time after j wakes up, and j delivers all the messages that were sent at least $\Delta + 2\Gamma + \Theta$ time after j wakes up.

6.4 Correctness

The Integrity and Total Order of Definition 2 for this algorithm follows directly from the correctness proof for the algorithm in the previous section. The only difference between the two algorithms is that when a process fails, some of its messages may be delivered by one correct process and not by another correct process. However, the method in which the messages are delivered (by slot, and in each slot according to processes indices) remains the same and the total order is still preserved. In addition, the following lemma proves the liveness condition.

Lemma 6.1 *All the messages received by a process i are delivered to the application.*

Proof: The only difference between the algorithm in this section and the one in the previous section is that if process i suspects j to be faulty in a slot s, i does not deliver j's messages in s. Thus, in order to prove the liveness property we need to show that if a process j sends messages to process i for a slot s, and j does not subsequently fail, then i does not consider j to be faulty in slot s. To prove this we show that (1) if at some point i does not suspect j and j does not subsequently fail, then i does not subsequently suspect j; and (2) if j recovers and re-joins the algorithm starting to send messages in slot s, then every correct process i will consider j to be correct at s and hence deliver j's messages for s.

(1) is true since i suspects j only if j's messages for s fail to reach i within $\Delta + \Gamma$ time from the time i finished s. From our assumptions on clock synchronization and network latency, this implies that j has failed. (2) is true since if j recovers and re-joins the algorithm, it sends a "join" message indicating the slot $Smax$ in which it will start sending messages. As argued above, $Smax$ is chosen to be late enough to guarantee that the "join" message arrives at all the correct processes before they start slot $Smax$. The processes process this message immediately upon receipt, and thus j is considered correct by the time it starts sending messages.

■

Note that in this proof we omit messages that are received during the initialization time. For these messages, we allow i not to deliver messages for slots that precede the slot i joins in.

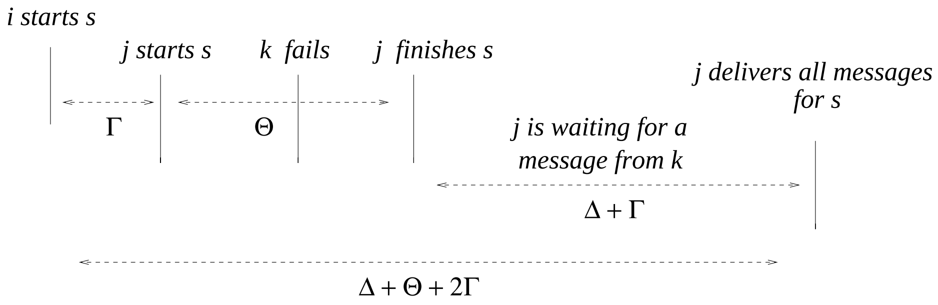
6.5 QoS guarantees

We first analyze the latency:

Theorem 5

$$AppLatency = \Delta + \Theta + 2\Gamma$$

Proof: As stated, a process j waits $\Delta + \Gamma$ time after it finishes slot s in order to receive all messages sent by all processes for this slot. If after that time a message from some process k did not arrive it knows (as proved in the correctness above) that k failed, and so after that time j finishes delivering messages for slot s. Let i and j be two correct processes, and assume i sends a message m in slot s. From our assumption on clock synchronization, we know that j finishes s at most $\Theta + \Gamma$ time after i starts slot s (see Figure 9). According to the algorithm, j delivers all messages sent for s at most $\Delta + \Gamma$ time after it finishes s. So j delivers message m at most $\Theta + \Gamma + \Delta + \Gamma = \Delta + 2\Gamma + \Theta$ time after the time it was sent by the application in i. Since the scenario in Figure 9 can actually occur, $\Delta + 2\Gamma + \Theta$ is the maximum delay for any message sent in this algorithm. ■



i starts s
j starts s
k fails
j finishes s
j delivers all messages for s
Γ
Θ
j is waiting for a message from k
Δ + Γ
Δ + Θ + 2Γ

Figure 9: Maximal delay when process failures are tolerated.

The transmission rate is not effected 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 and the rates sent in response to these messages. As mentioned before, we do not take the cost of initialization and renegotiation into account when analyzing the cost of the "normal" operation of the algorithm.

6.6 Notes on handling network partitions

When handling process joins, we have the join occur at the same point in the message sequence at all the processes including the joining process. This point defines the time starting which the liveness constraints hold, and the joining process has to deliver all the messages it receives. In contrast, for failed processes, we do not require that their failure be observed atomically (that is, at the same point in the message sequence). This is so since we implement total order with gaps.

When handling network partitions and merges, one has to decide whether to require that partitions and merges be observed atomically by all processes. If, like process failures, partitions do not have to be detected atomically, then the detached processes can be detected to be faulty using the existing mechanisms for detecting messages faults. If we do not require that merges be observed as occurring at the same point by all the processes, then handling merges can be done as follows: When a process i receives a message from a process j that is not in its active list of processes, i sends j a "merge" request for slot $Smax = (clock_j + \Delta + \Gamma)/\Theta + 1$ containing its current rate. When j receives this message, if i is in its active list it ignores the message. Otherwise, j treats it like a "join" message, that is, adds i to its list of active processes upon starting to deliver messages for slot $Smax$. In both cases, j sends a dual "merge" message with its own rate to i and tags the first message in slot $Smax$ with the slot number ($Smax$).

7 Message Loss

Using Definition 2 we can also relax the assumption that the network does not lose messages, and still preserve the total order. In this section we assume that messages may be lost, but there is a bound x on the number of consecutive messages sent by the same sender that the network can lose. This bound is one of the reserved QoS parameters, it is called *burst loss sensitivity* in [36][3].

We modify the algorithm of the previous section to accommodate message loss. To ensure that the ordering is computed consistently at all processes, we must make sure that all the messages containing rates will reach their destinations. To this end, each "init-rate" and "new-rate" message is sent $x+1$ consecutive times. To avoid wasting network resources, it is possible to append the rate to the first $x+1$ messages being sent at startup, after renegotiation and when the rate is sent to a new process that joins. Likewise, "join" messages have to be sent $x+1$ consecutive times. This mechanism makes sure that all the processes use the same ordering.

As before, when a process i is delivering messages from process j for a certain slot s, i moves to deliver messages from $j+1$ upon receipt of a dummy message or upon delivery of $AppMaxBurst$ messages from j. In addition, we must provide a mechanism for detecting lost messages. To this end, the slot number is included in the header of each message sent. When process i is delivering messages from process j for a certain slot s, i needs to be able to detect the fact that some messages that j sent for this slot were lost (or that the dummy message was lost in case less than $AppMaxBurst$ messages were sent in the slot), in order to proceed to deliver messages from $j+1$. Process i can detect this by one of two ways: First, if i waits $\Delta + \Gamma$ time from the time it finished slot s and it still did not receive all of j's messages for s, it knows it will never receive them. Second, if the next message from j pertains to slot $s+1$, then by the FIFO order no further messages for slot s may be received from j. As an optimization, when a process sends $AppMaxBurst$ messages in a slot, it can tag the last one with a bit denoting that it is the last message of the slot, to provide an additional way for detecting the end of the slot. In all cases, i proceeds to deliver messages from $j+1$.

However, if i times out on j, i cannot know whether j's message was lost, or whether j failed during this slot. Therefore, in this algorithm, i will keep j in the list of active processes and will still try to deliver messages from j during the following slots. Process i removes j from the active list only if it did not receive any messages from j during $x+1$ consecutive slots.

7.1 Correctness

In order to prove the correctness of the algorithm, we show that Definition 2 holds. Integrity follows from the fact that the network does not corrupt or spontaneously generate messages, and that our algorithm sends application messages at most once.

As for Total Order, the sending of rate messages $x+1$ consecutive times guarantees that processes' order information is kept consistent, and the ordering in each slot is preserved. The use of slot numbers in message headers makes sure that a message is delivered in the same slot by all the processes.

To prove Liveness, we must show that a processes does not detect the loss of a message that will later be received. Message loss is detected in two ways. First, using a time-out mechanism which was proven in the previous section to detect only cases in which messages are indeed lost.

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

Second, processes detect a lost message from j for a slot s when receiving a message from j for slot $s+1$. By the FIFO nature of the communication links, a message for slot s cannot be received after a message for slot $s+1$, so the loss is detected correctly and the Liveness specification is not violated.

7.2 QoS guarantees

The maximum latency remains the same as the delay in the previous section. This is because each time the application at process i changes its rate parameters we guarantee that all processes will receive the new rates by sending them $x+1$ times and thus all processes will use the right rates (as in the previous section) when delivering messages in each slot. In addition, we time out each slot in the same way we did in the previous section (each process waits at most $\Delta + \Gamma$ time after it finished slot s, and after that delivers all messages for that slot). Thus, the delay is unchanged.

The algorithm in this section does not send additional messages, but it does increase the transmission rate by adding the slot number at the header of each packet. The slot number can be represented by four bytes[4]. The significance of this addition depends on the payload message size. If application messages are fairly large, this addition is not significant.

The algorithm in this section also adds the transmission rate to the header of multiple messages. However, these messages are only sent at startup times, including startup of another process, (i.e., join) and renegotiation times. As mentioned before, we do not take the cost of initialization and renegotiation into account when analyzing the cost of the "normal" operation of the algorithm.

8 Leader Based Algorithms

In previous sections, we presented symmetric algorithms. We chose to consider symmetric algorithms, since it is difficult to make leader-based algorithms fault tolerant without violating QoS guarantees. When the leader fails, electing a surrogate leader may block the algorithm for an extensive period and induce significant delays. We do not discuss fault tolerant leader based algorithms in this paper. We only analyze the QoS guarantees of leader based algorithms in the absence of faults.

In this section we assume the VBR model of Section 5 (with no process failures or joins and no message loss), although we do not assume the process clocks are synchronized. We discuss two leader-based algorithms: a leader broadcast algorithm in Section 8.1, and a timestamp leader [6] in Section 8.2. We present general descriptions of how the algorithms work and study their QoS guarantees.

8.1 Leader broadcast algorithm

In the leader broadcast algorithm, all the processes send their messages to the leader. The leader sends the messages in the order they are received to all processes. When a process receives a message from the leader it delivers it to the application. Since the messages are sent from one process to all processes, and from the FIFO assumption on links, all the processes deliver messages in the same order. Each process has to reserve bandwidth only to the leader. The leader reserves bandwidth to all the processes.

In this algorithm each message is now sent twice, once from the sending process to the leader, and then from the leader to all processes. This causes a delay of 2Δ for a message to arrive at

[4] For slots of size of 100 milliseconds, four bytes will suffice for over three years.

all processes. In addition, messages are delayed at the leader, from the time the leader receives a message from the network and until it multicasts it. We denote this time by *processingOverhead*. The total delay is therefore:

$$AppLatency = 2\Delta + processingOverhead$$

Two points are worth noting about this latency. First, the processing overhead is not constant. If the network load is high, messages can pile up at the leader, and the delay until a message can be processed by the leader is bound to grow. This growth limits the scalability of leader based algorithms. When implementing a leader based algorithm, it is important to chose a powerful machine for the leader, in order to minimize the effect of this growth.

Second, it may be possible to chose a leader such that the network latency between the leader and every other processes (Δ) will be smaller than the latency between an arbitrary pair of processes (Δ in the previous sections). Since multicast is usually implemented using a rooted spanning tree, it may be possible to chose the leader to be close to the root of the spanning tree. However, the root itself is usually a router, and is not available to the application. Identifying a process with optimal latency to other processes is generally difficult.

The reserved transmission rate is increased by this algorithm, since every message is sent twice. First it is unicast to the leader and then the leader multicasts it to all processes.

Although unicast messages are in a sense "cheaper" than multicast messages as they are not propagated through the entire multicast tree, we cannot precisely compare the cost of a unicast message with the cost of a multicast message since this depends on the network topology.

8.2 Timestamp leader algorithm

In the timestamp leader algorithm, the leader does not multicast the messages themselves. Instead, whenever a process wants to send a certain number, k, of messages, it asks the leader for k timestamps and immediately multicasts the messages to all the processes. When the leader receives a request for k timestamps, it assigns the next k timestamps (or sequence numbers) to this process, and notifies all processes of this assignment. Each process delivers the messages according to the timestamp associated with them. For example, if a process i delivered a message with timestamp l, and timestamp $l+1$ is allocated to process j, then the next message i will deliver will be from j. If a process asked for k timestamps, there is a time limit on the time until it sends k messages. If it sees that by this time it does not have that many messages to send, it must send dummy messages instead. For the VBR model, this time limit may be chosen to be the slot size, Θ, and k may be chosen to be the *AppMaxBurst*.

Since the timestamps are assigned by a single process, they are consistent, and no two processes are assigned the same timestamp. By the FIFO order of links, delivering messages according to these timestamps guarantees total order.

In addition to the reservation for application messages, processes need to reserve bandwidth for sending timestamp requests and the leader must reserve extra bandwidth for the timestamp messages.

8.2.1 QoS guarantees

There are two cases in which a received message can be delayed before it can be delivered. First, a message can arrive prior to its assigned timestamp. Since a request for a timestamp is sent before the message is actually sent, the time it takes for the timestamp to arrive is at most $2\Delta +$

$processingOverhead$, Δ to arrive at the leader, $processingOverhead$ for the leader to respond to it and another Δ to arrive at each receiving process. Second, a message can arrive prior to other messages that have preceding timestamps. The time it takes until the preceding messages arrive is at most $\Delta + \Theta$ since after Θ time a process is obliged to send the number of messages it asked for. The total maximum delay is the maximum of these two delays, thus the total delay time after a message arrives is:

$$AppLatency = \Delta + Max(\Delta + processingOverhead, \Theta)$$

Note, that although we still take the leader processing time into account when analyzing the delay for this algorithm, this time will be much smaller because the leader only processes timestamp requests (one request is sent for k messages) as opposed to processing all the messages in the leader broadcast algorithm presented above.

This algorithm increases the transmission rate due to sending of timestamp requests and timestamps. These messages are sent once per slot. In addition, as in the algorithm of Section 5, the average transmission rate is increased by the sending of *dummy* messages, also sent at most once per slot. Thus, the average rate is increased by three messages per slot for each process. The maximum burst is only increased by two messages per slot for each process since we do not add a *dummmy* message in this case but the timestamp messages are still being sent.

9 Conclusions and Future Work

This paper sets the framework for analyzing QoS guarantees of totally ordered multicast services. Specifically, we have shown how to achieve totally ordered multicast without violating QoS guarantees in certain network models. We have shown that in failure prone networks one can achieve total order with gaps at a reasonable latency. Reliable total order, on the other hand, can induce latencies which are proportional to the number of faults occurring.

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 those suggested here for fast updates, and in the background implement reliable total 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 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 study is the cost of achieving reliability over an unreliable network. It would be interesting to study the latency/bandwidth tradeoffs between using retransmissions and using forward error correction (FEC) (for example, see [26, 35]) to achieve reliability.

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

[1] *10Six - A massive online team play over the Internet.* URL: http://www.tensix.com/ , http://www.heat.net/10sixchannel/index.html.

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

[3] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4), November 1995.

[4] The ATM Forum Technical Committee. *ATM User Network Interface (UNI) Specification Version 3.1*, June 1995. ISBN 0-13-393828-X.

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

[6] J. Chang and N. Maxemchunk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3), 1984.

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

[8] G. Coulson and G. Blair. Architectural Principles and Techniques for Distributed Multimedia Application Support in Operating Systems. *ACM Operating Systems Review*, 29(4):17–24, October 1995.

[9] Coulson, G. and Blair, G.S. and P. Robin. Micro-kernel Support for Continuous Media in Distributed Systems. In *Proceedings of Computer Networks and ISDN Systems 26*, pages 1323–1341, 1994.

[10] J. Crowcroft, M. Handley, and I. Wakeman. *Internetworking Multimedia.* UCL Press, September 1999. Available on-line from: http://www.cs.ucl.ac.uk/staff/j.crowcroft/mmbook/book/book.html.

[11] D. Dolev, S. Kramer, and D. Malki. Early delivery totally ordered broadcast in asynchronous environments. In *23rd IEEE Fault-Tolerant Computing Symposium (FTCS)*, pages 544–553, June 1993.

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

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

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

[15] L. Gautier and C. 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/.

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

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

[18] H.Q. Nguyen and C. Bac and G. Bernard. Integrating QoS Management in a micro-kernel based UNIX Operating System. In *Proceedings of the 23rd Euromicro Conference*, September 1-4 1997.

[19] IETF. *The Integrated Services (intserv) IETF Working Group Home Page, URL: http://www.ietf.org/html.charters/intserv-charter.html.*

[20] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Proceedings of the USENIX 1998 Annual Technical Conference*, June 1998.

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

[22] I. Keidar and D. Dolev. Totally ordered broadcast in the face of network partitions. *exploiting group communication for replication in partitionable networks.* In D. Avresky, editor, *Chapter 3 of Dependable Network Computing.* Kluwer Academic, 1999.

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

[24] C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic qos in real-time mach. *In the proceedings of Multimedia Japan 96*, april 1996.

[25] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 1997. http://www.cl.cam.ac.uk/Research/SRG/netos/nemesis/.

[26] J. B. M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *ACM SIGCOMM '98*, September 1998.

[27] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

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

[29] The MASH Project, University of California, Berkeley. *The MediaBoard.* URL: http://www-mash.cs.berkeley.edu/mash/projects/mboard/mb.html.

[30] 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.

[31] S. McCanne and V. Jacobson. Vic: A flexible framework for packet video. In *Proceedings of ACM Multimedia*, pages 511–522, November 1995.

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

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

[34] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Asynchronous fault-tolerant total ordering algorithms. *SIAM J. Comput.*, 22(4):727–750, August 1993.

[35] J. Nonnenmacher and E. W. Biersack. Reliable multicast: Where to use FEC. In *IFIP 5th International Workshop on Protocols for High Speed Networks (PfHSN'96)*, October 1996.

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

[37] L. Rodrigues, H. Fonseca, and P. Verissimo. Totally ordered multicast in large-scale systems. In *International Conference on Distributed Computing Systems (ICDCS)*, 1996.

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

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

[40] The Real-Time and Multimedia Laboratory in the Department of Computer Science at Carnegie Mellon University. *The Real-Time Mach Project.* URL: http://www.cs.cmu.edu/afs/cs/project/art-6/www/rtmach.html.

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

[42] 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.

[43] J. Wroclawski. *Specification of the Controlled-Load Network Element Service, RFC 2211*, September 1997. Internet Engineering Task Force, Network Working Group.

[44] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource reservation protocol. In *IEEE Network*, September 1993. The RSVP Project home page: `http://www.isi.edu/div7/rsvp/rsvp.html`.