An Adaptive Totally Ordered Multicast Protocol that Tolerates Partitions[*]

G. V. Chockler N. Huleihel
D. Dolev
Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem 91904, Israel
E-mail:{grishac,nabil,dolev}@cs.huji.ac.il,
http://www.cs.huji.ac.il/{~grishac,~nabil,~dolev}

Abstract

In this paper we present a novel algorithm that implements a totally ordered multicast primitive for a *Totally Ordered Group Communication Service (TO-GCS)*. TO-GCS is a powerful infrastructure for building distributed fault-tolerant applications, such as *totally ordered broadcast*, consistent object replication, distributed shared memory, Computer Supported Cooperative Work (CSCW) applications and distributed monitoring and display applications.

Our algorithm is *adaptive*, i.e., it is able to dynamically alter the message delivery order in response to changes in the transmission rates of the participating processes. This compensates for differences among participant transmission rates and therefore minimizes fluctuations in message delivery latency. Our algorithm is thus useful for soft real-time environments where sharp fluctuations in message delivery latency are not acceptable.

Our solution provides well-defined message ordering semantics. These semantics are preserved even in the face of site and communication link failures.

1 Introduction

A group communication service with a totally ordered multicast primitive, *Totally Ordered Group Communication Service (TO-GCS)*, is a powerful infrastructure for building distributed fault-tolerant applications. Some of these are *totally ordered broadcast* [1, 8, 10, 14, 12], consistent object replication [1, 12], distributed shared memory [8], Computer Supported Cooperative Work (CSCW) applications [18] and distributed monitoring and display applications [14]. Due to its importance for distributed computing, TO-GCS has inspired a great number of research projects in universities and research institutions world-wide. Isis [3], Horus [20], Totem [2, 16], Transis [7], Amoeba [11], RMP [22], Delta-4 [17] are only some of the systems that support TO-GCS.

[*]This work was supported by ARPA grant number 030-7310

In this paper we present a novel total ordering algorithm for TO-GCS. Our algorithm is *adaptive*: It is able to dynamically alter the message delivery order in response to changes in the transmission rates of the participating processes. This adaptation ability compensates for differences among participant transmission rates and thus minimizes fluctuations in message delivery latency. Many soft real-time applications make certain assumptions about message delivery latency, and therefore, sharp fluctuations in message delivery latency can wreak havoc in these cases. Our algorithm is thus useful for such applications.

Another important feature of our solution is that it provides well-defined message ordering semantics. These semantics are required by existing TO-GCS based applications [1, 8, 12] and are preserved in spite of both site and communication link failures. They were first formulated within the framework of the Extended Virtual Synchrony model [15] and elaborated in [8, 12, 21]. Further discussion of our algorithm's features appears in Section 1.2.

1.1 Problem Definition

A group communication service (GCS) classically consists of two main parts: a *membership service* and a set of *multicast services*. The task of the membership service is to maintain a listing of the currently active and connected processes and to deliver this information to the application whenever the membership changes. The output of the membership service is called a *view*. The multicast services deliver messages to the current view members.

The GCS multicast service suite typically consists of a set of primitives with different ordering/reliability guarantees. The most important among these is the *totally ordered* multicast service, which guarantees to deliver messages to the current view members in a consistent order. A GCS with a totally ordered multicast primitive is called a *Totally Ordered Group Communication Service(TO-GCS)*.

In this paper we concentrate on implementing an efficient totally ordered multicast service within the group communication framework. We assume that the underlying communication layer is represented by a basic view synchronous GCS that provides membership and FIFO multicast services. The minimal requirements of the underlying GCS appear in Section 2.

The principal correctness requirements imposed by our service are listed below. They are motivated by existing TO-GCS based applications [1, 8, 12]:

- A logical *timestamp* is attached to every message delivered by TO-GCS;
- The same timestamp is attached to a message at every process that delivers that message. This timestamp is unique system-wide and remains unique in face of network partitions;
- Every process delivers messages in the order of their timestamps;
- The timestamp order complies with the *global causal order on messages* [13], $\rightsquigarrow$, defined to be the reflexive transitive closure of the following:
  1. $m \rightsquigarrow m'$ if there exists a process p such that m was sent at p before m';
  2. $m \rightsquigarrow m'$ if there exists a process p such that m' was sent at p after m has been delivered at p.

Note that the above requirements imply that (1) any two messages are delivered in the same order at any process that delivers both of them, and (2) the message delivery order complies with the global causal order on messages.

In addition, the service implementation should satisfy the following liveness requirement:

- If a process p receives from the underlying communication layer an infinite number of messages from every operational and connected process, then p will eventually deliver every message supplied to it by the underlying communication layer or crash. (We further elaborate on TO-GCS liveness requirements in Section 3.2).

Note that the above problem is weaker in several ways than the well-known *Atomic Broadcast (AB)* problem found in the literature [10]. In particular, we do not require that each message multicast by a correct process will eventually be delivered by all correct processes; nor do we require that each message delivered by a correct process will be eventually delivered by all correct processes.

Our service is similar to the partitionable group communication service specified by the VS-machine of [8]. However, there are a few distinctions:

- Unlike [8], our service delivers application messages labeled with timestamps. The use of timestamps is motivated by the fact that TO-GCS with timestamps is useful for various TO-GCS based applications, e.g., it is utilized by the Consistent Object Replication Layer described in [12].
- The VS-machine of [8] provides safe indications, whereas TO-GCS does not. However, semantics similar to those achieved with safe indications can easily be achieved at the application level using end-to-end acknowledgments. This technique was demonstrated in [12].

1.2 Protocol Features

In this section we consider the main features of our total ordering protocol and discuss related work.

1.2.1 Dynamic Adaptation

Because a totally ordered multicast service is so useful, the efficiency of its implementation has become an important issue. A well-known technique for providing a totally ordered multicast delays delivery of a received message until the process has: (a) delivered all received messages which precede the message in question within the total order; and (b) learnt that every message that could preceded it will never arrive. This results in high latency in message delivery when not all the participant processes are uniformly active. Total ordering protocols which are based upon this technique are called *symmetric*. Another approach implemented by *sequencer* [3, 4, 11, 22] or *token* [16] based protocols uses extra messages (ordering messages or token requests) and is therefore less efficient under high loads [19].

The protocol presented in this paper is *dynamically adaptive*: Messages are assigned a wide range of priorities which are adjusted "on-the-fly" to reflect ongoing changes in process activities. Messages are then delivered in order of priority. The protocol testing results (see Section 6) show that after a short adaptation period the average message delivery latency incurred by our protocol is close to that of the underlying communication layer. Furthermore, the variance of the post-adaptation message delivery latency exhibited by our protocol is extremely low.

By contrast, under the same load patterns the latency incurred by traditional (non-adaptive) total ordering protocols is close to the transmission rate of the slowest process in the group. Moreover, these protocols exhibit sharp fluctuations in message delivery latency. This makes the message delivery latency incurred by such protocols much less predictable, causing problems for soft real-time applications. Our protocol is thus a solution for these problems.

Some systems differentiate between only two process activity levels. For example, [9] addresses the adaptivity issues by classifying group members as *active* or *passive* according to whether they have any messages to send or not; the right to multicast messages is then evenly distributed among all currently active processes. In the Hybrid protocol of [19], assignment of active or passive process status is based upon the relation between the process' transmission rate and the network delay: active processes run a symmetric protocol, while passive processes run a token-based one. Processes dynamically switch between active and passive states. The obvious limitation of the approach exemplified by these two protocols is that all the active (passive) processes are treated equally, while in practice it is rare that all of the active (passive) processes are uniformly active (passive).

In the ToTo protocol of [5] messages are delayed until messages are received from a *majority* of group members. ToTo achieves good latency only when: (a) the currently active members of the group form a majority, and (b) the processes that make up this majority broadcast their messages at approximately the same rates.

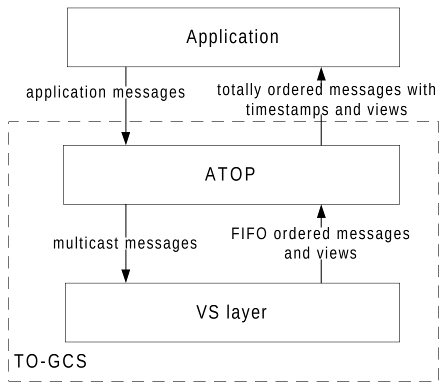

Application
application messages
totally ordered messages with timestamps and views
ATOP
multicast messages
FIFO ordered messages and views
VS layer
TO-GCS

Figure 1: The TO-GCS System Components and Interfaces

1.2.2 Partionable Semantics

Additional implementation challenges are raised by the fact that the service requirements stated in Section 1.1 should be satisfied in environments where multiple concurrent network components are allowed to coexist and where processes in each component are treated as non-faulty. With many existing group communication systems [3, 5, 11, 22] the following scenario is possible: Suppose that two processes disconnect from each other, while some common non-delivered messages remain in their buffers. If these messages' order has not yet been negotiated, they may either be delivered in an inconsistent order or be discarded. Either way, the service requirements are violated.

Special care is needed to prevent such situations from occurring. Common practice [16] is to attach some ordering information to each newly multicast message. This information should be sufficient to allow each process to consistently order the message so that the need to communicate with other processes is eliminated.

Things become more complicated if the message delivery flow is allowed to be dynamically adaptive. This is because message delivery order may be altered as a result of the adaptation decision. We must therefore be careful to preserve the message delivery semantics by guaranteeing the atomicity and uniformity of the adaptation decision. This is a main challenge of incorporating adaptation into a total ordering algorithm.

The technique presented in this paper allows various external adaptation policies to be combined with the total ordering protocol. The resulting multicast service combines two valuable features: suitable performance for soft real-time applications and sound partitionable semantics.

2 The Environment Model

We assume an asynchronous distributed environment. Further, we assume that processes can fail and restart, and that the network can partition into several disjoint components which can re-merge later on. The environment is equipped with a *view-synchronous group communication layer*, called the VS (View-Synchronous) layer. This layer guarantees reliable FIFO delivery of messages that have been multicast within a group of connected and active processes. Another VS layer objective is to provide *failure detection*: Possible changes in network connectivity and in failure status of the processes are relayed via special membership change reports, called *views*. The layer is called *view synchronous* because messages are guaranteed to be delivered in the view in which they were originated. Our *Adaptive Total Ordering Protocol (ATOP)* is built on top of the VS layer. The layer structure of the Totally Ordered Group Communication Service (TO-GCS) is depicted in Figure 1.

2.1 The VS Layer Guarantees

For the rest of this paper, we denote the following: P is a totally ordered finite set of processes; M is a message alphabet; $(I, <_I, i_0)$, a totally ordered set of view identifiers with initial view identifier i_0; $views = I \times 2^P$, the set of pairs consisting of a view identifier together with a set of processes; If $v \in views$, we write $v.id$ and $v.set$ to denote the view identifier and set components of v respectively.

We define the *current view* at a process p to be as follows: if the VS layer has delivered any views at p, then the current view at p is the last such view; otherwise, it is a pair consisting the distinguished initial view identifier i_0 and the process universe P. We say that a message m is sent (delivered) in a view v at p if m is sent (delivered) at p when the current view at p is v.

The VS layer is required to satisfy the following requirements:

View Identifier Identity: Views with different process sets have different identifiers.

Initial View Identifier Uniqueness: The identifier of any view delivered by the VS layer at any process differs from the initial view identifier i_0.

Local View Identifier Monotony: Views are delivered in the view identifier order at each process .

Self Inclusion: For any view v delivered by the VS layer at a process p, $p \in v.set$.

Message Integrity: For any message m delivered at a process p in a view v, there is a preceding send event at some process q. Moreover, m is sent in v at q.

No duplication: Every message delivered by the VS layer at a process p is delivered only once at p.

Reliable FIFO **Delivery:** For any two messages m, m', processes p, q, and a view v: If m is sent before m' in v and q delivers m', then q delivers m before m'.

3 The TO-GCS Specification

3.1 Correctness

In addition to the message ordering properties outlined in Section 1.1, we require that TO-GCS satisfies the following:

View Properties are similar to the first four guarantees provided by the VS layer (see Section 2.1);

Basic Message Delivery Properties:

1. For every message delivered by TO-GCS to the application there is a preceding send event. Furthermore, this send event occurs in a view whose identifier is not greater than that of the view in which the message is delivered;
2. Each message is delivered in the same view at any process at which the message is delivered;
3. The sender of a message is always a member of the view in which the message is delivered.

3.2 Liveness

We require of TO-GCS to satisfy the same liveness specifications as those guaranteed by the VS layer. Since the liveness specification of the VS layer is out of the scope of this paper (the interested reader may refer to [8, 21]), we only require that for every process p, ATOP at p preserves the liveness semantics provided by the underlying VS layer. More precisely, we require the following:

1. Every application message sent through ATOP is eventually transferred to the underlying VS layer unless a crash occurs;
2. If the VS layer delivers an infinite number of messages to ATOP from every member of the current view, then ATOP will eventually deliver every message that the VS layer has passed to it, or crash;
3. If the VS layer informs ATOP about a new view v, then ATOP will eventually deliver v or crash. Moreover, ATOP at p is bound (unless it crashes) to eventually deliver every message that the VS layer has transferred to it before v.

The first two properties together ensure that if all members of the current view keep sending messages, then ATOP at a process p preserves every liveness guarantee provided by the VS layer at network stability periods. For example: if, during the network stability periods, applications at all processes in the current stable component send infinite number of messages and the VS layer guarantees to deliver all messages sent through it (as required in [8]), then TO-GCS also guarantees to deliver every application message.

Note that the requirement that every process in the current view should issue an infinite number of messages may seem unrealistic. We require it only for the sole purpose of simplifying the protocol presentation. In the actual implementation, this precondition can be enforced by ATOP itself: it can simply multicast special *dummy* messages when its application becomes "silent" (see discussion in Section 5).

The third property ensures that if the VS layer provides additional liveness guarantees at times of new view installations, then ATOP will preserve them.

4 The Adaptive Total Ordering Protocol (ATOP)

In this section we describe the Adaptive Total Ordering Protocol (ATOP) which implements TO-GCS using the VS layer. The adaptive total ordering protocol at each process consists of two modules: the module implementing an adaptive total ordering *mechanism* (ATOM) and the module implementing an adaptation *policy* (AP). Such decoupling allows various external adaptation algorithms to be easily plugged into ATOP. The service structure is depicted in Figure 2.

4.1 The AP Module

The AP module is an implementation of an external adaptation policy. It thus keeps track of messages and views delivered by the VS layer, in order to learn about the transmission rate distribution among the current view members. From time to time (depending on the adaptation policy implemented) AP at p delivers a distribution, $dist$, to the ATOM module. A *distribution* is defined to be a pair consisting of: the *distribution identifier*, $dist.id$, taken out of a totally ordered set of distribution identifiers $(D, <_D, d_0)$ with an initial distribution identifier d_0; and a vector, called the *weights vector* and denoted $dist.w$, with an entry for each $q \in P$ such that $\sum_{q \in P} dist.w[q] = 1$.

Let v be the current view at a process p. We require that the following be satisfied by every distribution $dist$ delivered by AP at p in v:

- for each $q \in v.set$, $dist.w[q] \neq 0$, and for each $q \notin v.set$, $dist.w[q] = 0$;
- Let $dist'$ be a distribution delivered by AP at q in v. If $dist'.id = dist.id$, then $dist'.w = dist.w$. This means that every distribution delivered at any process in the same view has a unique identifier;
- $dist.id \neq d_0$.

4.2 Message Ordering Using Distributions

The ATOM module controls the message ordering using two distributions: the first one, called the *sending distribution*, is used to tag each newly multicast message; and the second one, called the *ordering distribution* is used to order incoming messages. In Section 4.3 we describe in detail how these distributions are maintained.

In addition, ATOM at each process has a copy of a pre-defined pseudo-random number generator G. This generator along with the current ordering distribution's weights vector fix a deterministic sequence of process identifiers. Messages tagged with a distribution

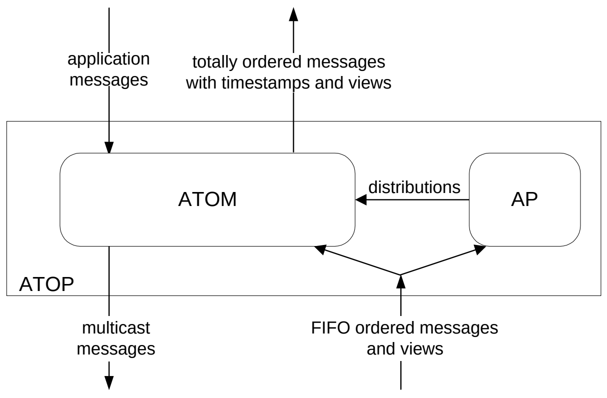

application messages
totally ordered messages with timestamps and views
ATOM
distributions
AP
ATOP
multicast messages
FIFO ordered messages and views

Figure 2: The ATOP Implementation

$dist$ are delivered when the current ordering distribution is equal to $dist$ and ordered by the sequence generated by G and $dist.w$.

This is illustrated by the following example: Let processes p, q and r be members of the current view. Assume that the sequence produced by G and the current ordering distribution's weight vector starts with $p, q, p, q, p, p, r, p, \ldots$. (Note that in this sequence, p apparently has a larger weight than r).

According to this sequence the first slot in the total order for the current view should be reserved for p's message. p's turn will be skipped only if the next pending p's messages is tagged with a different distribution; or if there are no more undelivered messages sent by p and there is a new view delivered by the VS layer. Likewise, the second slot in the total order should be reserved for q's message, the third slot again for p's message, and so on. Thus, for each slot, the protocol either waits for a message from the appropriate process or guarantees that no such message can be delivered in this slot, in which case it is skipped.

A detailed desctiption of the totally ordered delivery algotrithm is given in the next section.

4.3 The ATOM Module

The responsibilities of the ATOM module are as follows:

- To associate application messages with distributions;
- To guarantee that the delivery order of messages is determined by the distributions associated with them;
- To preserve the semantics provided by the underlying VS layer (see Section 2.1) in all aspects concerned with view delivery, the relative order of messages and views, and the reliable FIFO order within each view. This facilitates the achievement of the View and Basic Message Delivery properties of TO-GCS (outlined in Section 3);
- To extend the VS layer's FIFO delivery order within each view to the total delivery order in that view, so that the ordering semantics of Section 1.1 are satisfied.

ATOM associates distributions with messages by giving each newly multicast message a tag consisting of the maximal identifier among all distributions known to this instance of ATOM. The AP modules at participating processes must therefore make sure that the distribution with the maximal identifier corresponds to the most recent process transmission rate distribution. The adaptation policy described in Section 6 achieves this by allowing the AP module at only a single process (within the current view) to inject new distributions. This process is chosen deterministically from among the current view members. Other possible ways to implement the AP module are discussed in Section 7.

Within each view, message delivery order is determined by the distribution with the minimal identifier among all distributions attached to this view's yet-undelivered messages. This distribution is called the *ordering distribution*.

ATOM guarantees that the next message to be delivered is tagged with the current ordering distribution identifier. Since the AP module guarantees that every distribution has a unique identifier within each view (see Section 4.1), this means that the delivery order of each message is determined by the same distribution at any process that delivers this message.

Thus, within each view, every message is delivered in the same order at all processes that deliver this message (even at those processes that may become disconnected). Furthermore, since views are delivered in the same order at every process and each message is always delivered in the same view, ATOM guarantees the *global* total delivery order of messages.

The message timestamp assigned by ATOM is thus a triple consisting of the identifier of the view in which the message is delivered, the identifier of the current ordering distribution and the sequence number of the message within the current ordering distribution. We can easily see that the timestamp assigned in such a way satisfies the TO-GCS ordering requirements (see Section 1.1):

1. The timestamp is globally unique because: (1) the VS layer View Identifier Identity property guarantees that each view has a unique identifier, and (2) every distribution has a unique identifier within each view;
2. Each message has the same timestamp at every process that delivers this message because: (1) each such process delivers the message in the same view; (2) within each view, the message is assigned the distribution identifier when it is initially

sent; and (3) within each view, messages which are stamped with the same distribution identifier, are delivered in the same order;

3. Messages are delivered in the order of their timestamps because: (1) the VS layer Local View Identifier Monotony property ensures that views are delivered in the order of their identifiers, and (2) the identifier of the current ordering distribution increases monotonically within each view;

4. Each message m' sent by p after the delivery of another message m cannot be delivered before m at p. Therefore, when m' is delivered it is given a timestamp greater than that of m. Since ATOM preserves the VS layer's FIFO delivery order, thus the timestamp order satisfies causality.

Finally, to satisfy the TO-GCS liveness requirements, ATOM should not arbitrarily deliver any new views, nor should it arbitrarily change the current ordering distribution: instead, ATOM may deliver a new view only after it has validated that no more new messages belonging to the last view delivered to the application will ever arrive, and it may change the current ordering distribution only after it validates that no more messages tagged with the current ordering distribution will ever be received in the current view.

If this is not observed, then the following situation is liable to arise: suppose that a message m belonging to a view v were to arrive after a newer view had been delivered to the application. In this situation, correctness can only be preserved if we discard m. This, however, violates the liveness requirements. A similar situation will occur if a message tagged with some distribution identifier arrives after the current ordering distribution has been reset to a newer value.

We utilize the VS layer's Message Integrity property in order to tell that no more messages will be delivered by the VS layer in some view. This property implies that after the VS layer delivers a view v, it will not deliver any message sent in any view delivered before v in the future.

We make use of the VS layer's Message Integrity and Reliable FIFO delivery properties in order to verify that no more messages tagged with some distribution identifier d will be delivered in the current view. These properties imply that: if for each member of the current view the VS layer has delivered either (1) some message tagged with a distribution identifier greater than d or (2) a new view, then the VS layer will never deliver any message stamped with d within the current view.

A detailed description of the ATOM module algorithm is given below.

Sending Messages

The ATOM module at a process p learns about new distributions either directly from p's AP or from messages sent by ATOMs at other processes.

Let v be the current view at p. We define the *sending distribution* at p to be the distribution with the maximal identifier among distributions received at p in v, if any, otherwise it is a distribution $dist_v$, called the *default distribution for* v, such that $dist_v.id = d_0$ and $dist_v.w[q] = 1/|v|$ for each $q \in v.set$, and $dist_v.w[q] = 0$ otherwise.

The following attributes are attached by the ATOM module at a process p to each newly multicast application message:

- *sender*: the p's identifier;
- *dist_id*: the identifier of the current sending distribution at p;
- *seqno*: the sequence number of this message within the current sending distribution at p.

The first message to be tagged with a distribution's identifier will also bear the weights vector component of this distribution. (This is in addition to the above mentioned values.)

Basic Message and View Delivery

ATOM buffers messages and views delivered by the VS layer. Views are stored in the set $PendingViews$. Messages delivered by the VS layer in a view v are stored in the set $PendingMsgs[v.id]$.

Views delivered by ATOM are taken from the $PendingViews$ set. The next view chosen for delivery is the view with the minimal identifier among the views currently in the $PendingViews$ set. Delivered views are deleted from the $PendingViews$ set. Obviously, since the VS layer guarantees to deliver views in the order of their identifiers, the same is true for ATOM as well.

If v is the last view delivered by ATOM to the application, then the next message to be delivered is taken from the $PendingMsgs[v.id]$ set. ATOM suspends delivery of new views as well as of messages sent in these views until both conditions shown in Figure 3 are true.

1. $PendingViews \neq \emptyset$;
2. $PendingMsgs[v.id] = \emptyset$;

Figure 3: The Conditions for the New View Delivery

Since the VS layer guarantees that each message m is delivered in the same view at every process that delivers m, then there exists a view identifier i such that $m \in PendingMsgs[i]$ at any process that received m from the VS layer. Thus each ATOM module that delivers m to the application, delivers m in the same view.

Message Delivery within a View

Let Num be an enumeration of process identifiers in P. Let w be a weights vector as defined in Section 4.1 and G_w be a pseudo-random number generator which produces *Num*(r) with probability $w[r]$ on each invocation. We denote a pseudo-random number obtained on the ith invocation of G_w by $G_w(i)$, $i \geq 0$.

The ATOM module at each process p maintains the following data structures:

- *ordDist* holds the current ordering distribution. It is initialized to be the default distribution for (i_0, P);

- $next[q]$ is the total number of messages sent by q which have been ordered by $ordDist$. Initially, $next[q]$ is set to be 0 for each $q \in P$.

  Clearly, the sum of $next[q]$ for each $q \in P$ holds the total number of messages that have been ordered by $ordDist$ so far. We define $totalOrdered \stackrel{\text{def}}{=} \sum_{q \in P} next[q]$.

- *TS* is a timestamp attached to every message delivered to the application. At any state of the protocol TS is defined to be a triple consisting of the current view identifier, $ordDist.id$ and $totalOrdered$ variables. The order on timestamps is lexicographic;

- *G* is an instance of a pseudo-random number generator known by all processes in P. Initially, G is set to be $G_{dist_{(i_0,P)}.w}$ and is initialized to some predefined seed agreed upon by all processes in P.

Let v be the view which was most recently delivered by ATOM to an application at a process p, if any, or (i_0, P) otherwise. We first consider how $ordDist$ is maintained. $ordDist$ is initially set to be the default distribution for (i_0, P). Whenever ATOM delivers a new view to the application $ordDist$ is set to be the default distribution for this view. Within each view $ordDist$ is reassigned a new distribution when all the conditions depicted in Figure 4 are true.

1. $PendingMsgs[v.id]$ does not contain any message tagged with $ordDist.id$;
2. There is some message $m \in PendingMsgs[v.id]$ such that $m.dist\_id > ordDist.id$;
3. For each $q \in v.set$, there is a message m sent by q such that $m.dist\_id > ordDist.id$, or $PendingViews \neq \emptyset$.

Figure 4: The Conditions for Changing the Ordering Distribution

Whenever the value of $ordDist$ changes (as a result of either a new view delivery or fulfillment of conditions in Figure 4) the following steps are performed:

1. $ordDist$ is assigned that distribution whose identifier is minimal from among the identifiers of all distributions attached to the messages currently in $PendingMsgs[v.id]$. Note that because the VS layer guarantees the reliable FIFO delivery within a view, there is always a message in $PendingMsgs[v.id]$ which contains the new distribution's weights vector;
2. $next[q]$ is set to 0 for each $q \in P$;
3. G is set to be $G_{ordDist.w}$ and is initialized to some seed agreed upon by all members of v.

The ATOM module delivers a pair (m, TS) on the next invocation of its delivery procedure iff the conditions sketched in Figure 5 are satisfied. Note that these conditions imply that (1) the next message to be delivered (from among messages currently in $PendingMsgs[v.id]$) is determined according to the weights vector of the distribution in which this message was sent; and (2) the message delivery order is consistent with the order of message sending, i.e., the message delivery order preserves FIFO.

1. $m \in PendingMsgs[v.id]$;
2. m is tagged by $ordDist.id$;
3. m is sent by a process q such that $G_{ordDist.w}(totalDelivered) = Num(q)$;
4. $m.seqno = next[q]$.

Figure 5: The Conditions for the Delivery of (m, TS)

Whenever a pair (m, TS) is delivered to the application the following steps are performed:

1. m is removed from $PendingMsgs[v.id]$;
2. $next[m.sender]$ is incremented;

If none of the conditions in Figures 3, 4 and 5 are satisfied, ATOM blocks unless the conditions in Figure 6 are true. These conditions indicate that no more new messages stamped with $ordDist.id$, which were sent by a process q such that $G_{ordDist.w}(totalDelivered) = Num(q)$ in v, will ever be received from the VS layer. We can therefore try to deliver another message in $PendingMsgs[v.id]$ labeled with $ordDist.id$ (if such a message exists).

1. $PendingMsgs[v.id]$ does not contain any message m sent by a process q such that $G_{ordDist.w}(totalDelivered) = Num(q)$;
2. There is a message $m' \in PendingMsgs[v.id]$ sent by q such that $m'.dist\_id > ordDist.id$, or $PendingViews \neq \emptyset$;
3. There is another message in $PendingMsgs[v.id]$ labeled with $ordDist.id$.

Figure 6: The Conditions for Skipping the Current Timestamp

In this case, ATOM increments $next[q]$, and thus *skips* a message that could have been sent by a process q and tagged by the current values of $ordDist.id$ and $next[q]$. This way other messages in $PendingMsgs[v.id]$ which are stamped with $ordDist.id$ and have not yet been delivered, get a chance to be delivered in one of the successive delivery attempts.

Finally, if all the conditions in Figures 3, 4, 5 and 6 are false, ATOM blocks until the VS layer delivers either a new message or a new view, which would in turn cause one of the aforementioned conditions to become true.

5 Remarks on the ATOP Performance Guarantees

There are two important issues that were intentionally left out of consideration in the ATOP protocol definition in the previous section: They are *flow control* and *failure detection*. This simplification allowed us to better concentrate on subtleties of achieving adaptive total ordering in partitionable environments. However, this inevitably weakened our performance claims.

For example, since changing the ordering distribution requires a message tagged with a greater distribution identifier from each member of the current view (see Figure 4), a single process (or a group of processes) that has no application messages to send may substantially slow down switching to the new ordering distribution. In this case, an appropriate flow control mechanism will enforce each such "silent" process to issue a *dummy* message tagged with a new distribution identifier, thus speeding up the agreement.

Unfortunately, in distributed asynchronous systems with failures there are situations in which no flow control algorithm can help much. In particular, the performance of ATOP depends in great extent on how fast the underlying VS layer delivers messages and how fast faulty processes are removed from the view. For example, if the VS layer fails to guarantee timeliness of failure detection, the ATOP protocol may be subject for significant delays during network instability periods.

The above problems can be addressed by combining the adaptation policy, failure detection and flow control mechanisms together within the same module. For example, the failure detector can use distributions produced by the AP module to guarantee that each process either transmits messages in the rate corresponding to its weight or is taken out of the current view.

Thus, for example, an instance of the failure detector at a process p will take care of situations in which p has no application messages to send by issuing special *dummy* messages in the rate corresponding to the current p's weight. Subsequently, if an instance of the failure detector at q fails to hear messages from p in the rate which roughly corresponds to the current p's weight it will suspect p and initiate the view change. Note, that since the adaptation policy is based on application messages (and not on *dummy* ones), this mechanism would not affect the adaptation decision.

6 The ATOP Implementation and Performance Results

In order to evaluate the performance of ATOP, we implemented a simple adaptation policy. This is described in Section 6.1 below. The resulting protocol was implemented over the Causal Multicast Service (CMS) of the Transis GCS [7] which satisfies the VS layer correctness specifications presented in Section 2.

6.1 An Adaptation Policy Implementation

In the adaptation policy we implemented, only the AP module at a single process deterministically chosen among the current view members, called a *book-keeper*, has the right to inject new distributions. The book-keeper's algorithm is as follows.

Let v be the current book-keeper's view. The book-keeper maintains a *sliding window* of messages delivered by the VS layer in v. The size of the window is $N \cdot |v.set|$, where $N \in \mathcal{N}^{>0}$ is the protocol's parameter called the *window size factor*. Let $N[r]$ denote the number of messages sent by a process r from among the messages currently in the sliding window.

Let ϵ be a small positive real number. The book-keeper maintains a vector, $weights$, with an entry for every process in P such that at any instant, $weights[r] = (N[r]+\epsilon)/|v.set|(N+\epsilon)$ if $r \in v.set$, and 0 otherwise. Thus, the $weights$ vector approximates the distribution of the process transmission rates among the members of $v.set$. The parameter ϵ is needed to avoid assignment of zero weights to the $v.set$ members. Note that $\sum_{r \in v.set} weights[r] = 1$ at all times.

The $dist\_no$ variable counts distributions that have been output in the current view. Whenever a new distribution is output, $dist\_no$ is incremented and the content of the $weights$ vector is saved in another vector called $last\_weights$. If no distribution has yet been produced, $last\_weights[r] = 1/|v.set|$ for each $r \in v.set$, and 0 otherwise. Periodically, the distributions stored in $weights$ and $last\_weights$ are compared. If the difference between these distributions exceeds a predefined threshold, a distribution $dist$ such that $dist.id = dist\_no$ and $dist.w = weights$ is output.

6.2 The Testing Environment

We tested our protocol on 6 *Pentium-120* machines running the *BSD/OS* operating system and connected by a 10 MBit/second Ethernet LAN. Of these, two machines multicast at a rate of approximately 10 messages/sec and 1 machine that multicast at a rate approximately 20 messages/sec. The remaining 3 machines multicast at substantially lower rate which varied from one experiment to another. The message size was 50 bytes. During the testing period all the machines were connected and active. The observed message loss was negligible.

A potential weakness of this testing environment is that the transmission rates of participating machines was preset in advance and was static during each experiment. In the future we intend to analyze the performance of our protocol in more dynamic settings (see Section 7).

6.2.1 Performance Results Analysis

In our experiments we compared the performance of ATOP with a non-adaptive symmetric total ordering protocol, *All-Ack* [6], as well as with the Transis CMS. The Transis CMS guarantees only that the message delivery order satisfies the causal partial order on messages. Thus, in the Transis CMS, message order should not be agreed upon by all processes before delivery. Therefore, The Transis CMS (in the absence of message loss) has an average message delivery latency close to that of the underlying network, as well as a low message delivery latency variance. We thus chose the results of the Transis CMS message delivery latency analysis as references for the best achievable by any total ordering protocol.

In the first experiment series, we ran the All-Ack, ATOP and Transis CMS protocols while in each new experiment the transmission rate of each slow machine was smaller than it was in the previous experiment. We observed that (1) in each experiment the average message delivery latencies incurred by ATOP after adaptation and Transis CMS were close to one another (the average latency of ATOP was slightly greater); and (2) the latency of the All-Ack protocol steadily increased from one experiment to another.

The conclusion is that the average latency of All-Ack, unlike that of ATOP, varies according to the transmission rate of the slowest process and therefore cannot be predicted in advance.

In the second group of experiments we fixed the transmission rate of each of the slow processes to be approximately 1 message every 3 sec, and measured the variance in the message delivery latency. To do this, we compared the delivering rate of messages sent by a particular process, with the sending rate of those same messages. We observed for each of the tested protocols, that while the average delivery rate for messages sent by each process is close to the transmission rate for that process, the message delivery rate variance of All-Ack (see Figure 7(b)) is much greater than that of ATOP (after the adaptation) (see Figure 7(a)).

Figure 8(b) illustrates that in the All-Ack protocol, the message delivery blocks until a message from the slowest process arrives. Then, all pending messages are delivered at once. By contrast the post-adaptation message delivery rate of ATOP (see Figure 8(a)) is almost constant and close to the sender's transmission rate.

7 Other Adaptation Policies

The adaptation policy described in Section 6 is suited for LAN environments, where all connected processes see more or less the same picture. The same is not true for wide area networks, where the variance in the message round trip time among different processes might be significant, and different processes do not necessarily observe the same distribution for process transmission rates. Here, it is not a good idea to give the book-keeping responsibilities to a deterministically chosen process.

A better adaptation policy would instead dynamically reassign book-keeping responsibilities, while taking into account inter-process round trip delay times. The process with the minimal variance of inter-process round trip delays would obviously be the best candidate for current book-keeper.

Further challenges are presented by scenarios in which one or more participating processes may occasionally pause and then resume communication before being taken out of the current view. Clearly, such perturbing processes can easily cause the adaptation policy of Section 6 to not stabilize. An adaptation policy that would result in better performance would thus identify such perturbing processes and assign them weights which would rapidly decrease the influence of their past transmission activity (e.g., one can use the exponential backoff technique).

A completely different approach is to make the adaptation policy application dependent. For example, the application can specify possible message transmission rate distribution patterns in advance. The adaptation policy can thus recognize these patterns earlier and correspondingly change the current ordering distribution. In particular, this is useful in environments where message transmission rate distribution pattern depends on the time of day.

Acknowledgments

We are thankful to Dahlia Malkhi, Gil Neiger, Yehuda Afek and anonymous referees for their valuable comments which helped us to substantially improve the presentation quality. Idit Keidar greatly contributed to this work with her original ideas and insight. We would like to thank Tal Anker and Ohad Rodeh for suggesting interesting testing ideas. We are grateful to Aviva Dayan for proof reading our drafts and helping to improve the overall presentation quality.

References

[1] AMIR, Y. *Replication Using Group Communication Over a Partitioned Network*. PhD thesis, Institute of Computer Science, The Hebrow University of Jerusalem, Israel, 1995.

[2] AMIR, Y., MOSER, L. E., MELLIAR-SMITH, P. M., AGARWAL, D. A., AND CIARFELLA, P. The totem single-ring ordering and membership protocol. *ACM Trans. Comp. Syst. 13*, 4 (November 1995).

[3] BIRMAN, K. P., SCHIPER, A., AND STEPHENSON, P. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comp. Syst. 9*, 3 (1991), 272–314.

[4] CHANG, J. M., AND MAXEMCHUCK, N. Realiable Broadcast Protocols. *ACM Trans. Comput. Syst. 2*, 3 (August 1984), 251–273.

[5] DOLEV, D., KRAMER, S., AND MALKI, D. Early Delivery Totally Ordered Broadcast in Asynchronous Environments. In *23rd Annual International Symposium on Fault-Tolerant Computing* (June 1993), pp. 544–553.

[6] DOLEV, D., AND MALKI, D. The design of the transis system. In *Theory and Practice in Distributed Systems: International Workshop* (1995), K. P. Birman, F. Mattern, and A. Schipper, Eds., Springer, pp. 83–98. Lecture Notes in Computer Science 938.

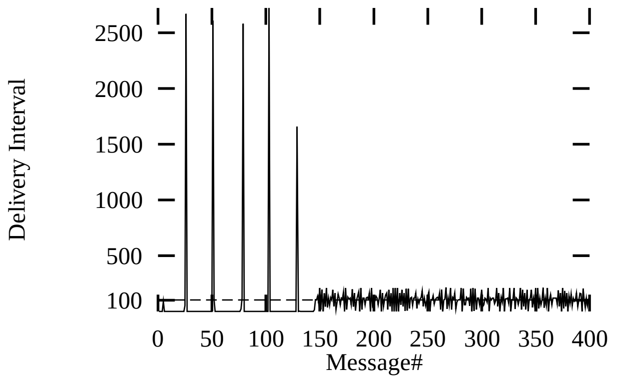
[7] DOLEV, D., AND MALKI, D. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM 39*, 4 (April 1996).

[8] FEKETE, A., LYNCH, N., AND SHVARTSMAN, A. Specifying and Using a Partionable Group Communication Service. In *16th Annual ACM Symposium on Principles of Distributed Computing* (August 1997).

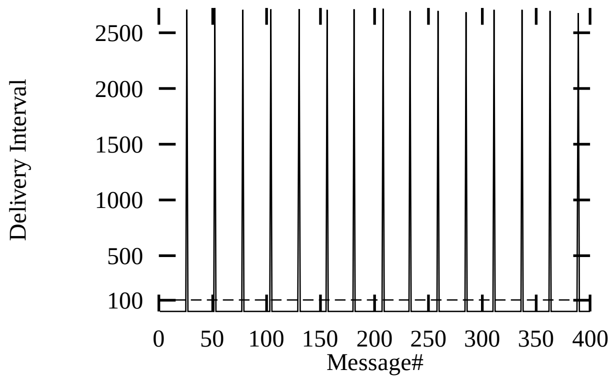
[9] FRIEDMAN, R., AND VAN RENESSE, R. Packing Messages as a Tool for Boosting the Perfomance of Total Ordering Protocols. TR 95-1527, dept. of Computer Science, Cornell University, August 1995.

[10] HADZILACOS, V., AND TOUEG, S. Fault-Tolerant Broadcasts and Related Problems. In *chapter in: Distributed Systems*, S. Mullender, Ed. ACM Press, 1993.

[11] KAASHOEK, M. F., AND TANENBAUM, A. S. An evaluation of the Amoeba group communication system. In *Proceedings of the 16th International Conference on Distributed Computing Systems* (May 1996), pp. 436–447.

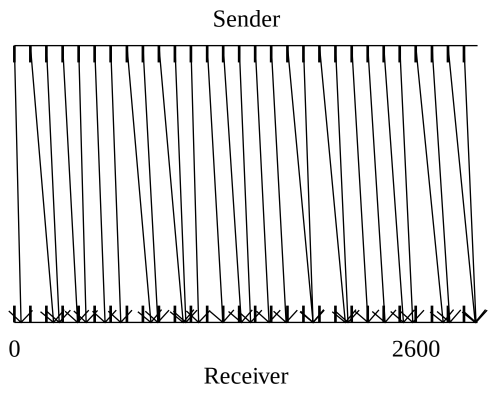


(a) The Post-Adaptation Delivery Rate of ATOP

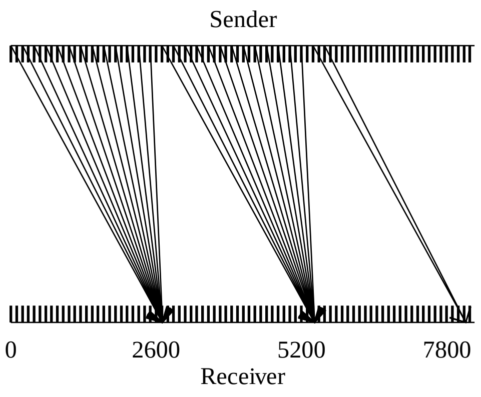


(b) The Delivery Rate of All-Ack

Figure 7: Delivery Rates for Messages Sent at ∼10 msgs/sec



(a) ATOP After Adaptation



(b) All-Ack

Figure 8: Message Delivery Rates vs Transmission Rates for Messages Sent at ∼10 messages/sec ($1\ tick \sim 100\ msec$)

[12] KEIDAR, I., AND DOLEV, D. Efficient Message Ordering in Dynamic Networks. In *15th Annual ACM Symposium on Principles of Distributed Computing* (May 1996), pp. 68–77.

[13] LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM 21*, 7 (July 78), 558–565.

[14] MALKI, D. *Multicast Communication for High Avalaibility*. PhD thesis, Institute of Computer Science, The Hebrow University of Jerusalem, Israel, 1994.

[15] MOSER, L. E., AMIR, Y., MELLIAR-SMITH, P. M., AND AGARWAL, D. A. Extended Virtual Synchrony. In *Intl. Conference on Distributed Computing Systems* (June 1994). Also available as technical report ECE93-22, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA.

[16] MOSER, L. E., MELLIAR-SMITH, P. M., AGARWAL, D. A., BUDHIA, R. K., AND LINGLEY-PAPADOPOULOS, C. A. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM 39*, 4 (April 1996).

[17] POWELL, D. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[18] RODDEN, T. A survey of CSCW systems. *Interacting with Computers 3*, 3 (1991), 319–353.

[19] RODRIGUES, L. E. T., FONSECA, H., AND VERISSIMO, P. Totally ordered multicast in large-scale systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems* (May 1996), pp. 503–510.

[20] VAN RENESSE, R., BIRMAN, K. P., AND MAFFEIS, S. Horus: A Flexible Group Communication System. *Communications of the ACM 39*, 4 (April 1996).

[21] VITENBERG, R., KEIDAR, I., CHOCKLER, G. V., AND DOLEV, D. Group Communication System Specifications: A Comprehensive Study. Tech. rep., Inst. of Comp. Sci., The Hebrew University of Jerusalem, 1997. In preparation.

[22] WHETTEN, B., MONTGOMERY, T., AND KAPLAN, S. A high perfomance totally ordered multicast protocol. In *Theory and Practice in Distributed Systems: International Workshop* (1995), K. P. Birman, F. Mattern, and A. Schipper, Eds., Springer, pp. 33–57. Lecture Notes in Computer Science 938.