TOMP
A Total Ordering Multicast Protocol

Mahmoud DASSER
Laboratoire d'Iformatique Technique
EPFL - EL Ecublens. CH-1015 Lausanne
Switzerland

email : pleinevaux@litsun.epfl.ch (EARN/BITNET)
C=CH;A=ARCOM;P=SWICH;O=EPFL;OU=LITSUN;S=PLEINEVAUX (X400)

1. INTRODUCTION

A Reliable Multicast Transport layer (connection oriented or connection-less oriented) must provide two major, well-defined classes of functions. The first one called the *Transmission Function*, must provide a service that ensures message transmission from one host to a set of hosts belonging to a Multicast Group. The Multicast Group is assumed to be known in advance. The second class, called the *Reliable Multicast Function*, concerns the quality of service, the primitives provided to the user layer and the expected level of reliability. As opposed to the first class, the functions we find here depend on the aims of applications and their environment constraints. Birman & Joseph [BirJos88] had established a classification of the Reliable Multicast Functions. They assumed that a reliable service of the data transmission is provided by the lower layer. (In the case of ISIS system [Birman88], the reliable transfer service is provided by the TCP/IP layers or by UDP/IP layers and an extra module.). The 3 classes defined in [BirJos88] are the following:

(1) Atomic Broadcast Protocols: the Atomicity is one of the major problems in the Multicast protocols. Atomic multicasting means that if a message is sent to N hosts, it must be received by all the N hosts or none. This was solved by protocols based on two-phase commit algorithms [Gray 78].

(2) Ordered Broadcast Protocols: for some distributed applications the order in which messages are received is critical and must be the same at all hosts, even though this order is not determined in advance. Among the protocols that provide this service are the *ABCAST* protocol described in [BirJos87] and the protocol proposed by [Chang84].

(3) Causal Broadcast Protocols: this property was described formally in [Lamport78] and [BirJos88]. Informally we can define this property as follows: if message *A* causes the transmission of message *B;* then message *A* must be delivered everywhere before message *B*. Message *A* is said to be *causal* to message *B*. An example of a protocol implementing the causal property is *CBCAST*. It is used in the ISIS system and described in [BirJos87a]. Another Causal Broadcast Protocols protocol was proposed by [SCHIPER89].

Among the weak points of the existing reliable multicast protocols we note:

- a great number of messages exchanged between the group members to achieve the execution of a given protocol. For example, in the *ABCAST*, the multicast of one message causes the transmission of *3N* messages if *N* is the number of members in the group.
- the *latency time*, meaning the time expired from the moment a hosts receives a message to the moment it effectively delivers this message to the user.

In this paper we first briefly present the protocol implementing the ABCAST primitive in the ISIS system. We then show that this protocol can be enhanced, with little effort, to reduce the latency time of messages. Finally, we describe our protocol and the way it enhances the performance of the ABCAST protocol in more detail.We have simulated the two algorithms (the algorithm used in ABCAST and our new solution) on a SUN3/60 system. The simulation shows approximately 30-40% reduction of the latency time by the new solution in relation to the ABCAST protocol.

2. DESCRIPTION OF THE ABCAST PROTOCOL

The ABCAST Protocol, proposed by Dale Skeen and described under the ABCAST name in [BirJos87], ensures both the atomicity of the multicast messages and their total ordering. The atomicity property is ensured by the two-phase commit [Gray78] mechanism, whereas the total ordering property is based on a mechanism of timestamping messages as described below.

2.1. PROTOCOL

As mentioned above, this protocol runs in two distinct phases: the first is the dissemination phase, and the second is the decision phase.

2.1.1. DISSEMINATION PHASE

The emitter host multicasts a message to a group of receivers. When a host, i.e. a member of the group, receives a new message, it gives it a unique timestamp, puts it in the reception queue (`R_Queue`), and labels it undeliverable (*ud*). Afterwards, the receiver host sends an acknowledgement message to the sender with the timestamp assigned to this new message (`Local_Timestamp`).The sender collects all the proposed timestamps included in the acknowledgement messages sent by each host. The timestamp is a local sequence number incremented after each new message reception. To ensure the uniqueness of timestamps within the group, each host postfixes the sequence number with its host identifier. e.g. `host1` timestamps incoming messages with the sequence 1.1, 2.1 and so on.

2.1.2. DECISION PHASE:

As soon as the emitter host receives all the acknowledgements from all the hosts, it multicasts a `'validation message'` including the greatest value of the proposed timestamps which will be the `'final timestamp'`. When a host receives the validation message, it assigns the `final timestamp` to the corresponding message and labels it as a `deliverable` (*dl*) message. Then the receiver host orders the pending messages (in its reception queue), in ascending order of timestamps. At a given host, a message is delivered to the upper layer if and only if it verifies the following two conditions:

(1) it must be labelled as deliverable (*dl*);

(2) it must be at the head of the reception queue (having the lowest timestamp of the waiting messages in the reception queue).

We shall not discuss what happens if a failure occurs here. For example, if the sender does not receive the acknowledgement message from a particular host; this is done in [BirJos87].

2.2. ABCAST: ILLUSTRATION BY AN EXAMPLE

The figure below illustrates an example used in [BirJos88]. It concerns a group of 3 hosts: `host1, host2` and `host3`. Each host multicasts messages *m1, m2* and *m3* respectively to the same group of hosts. We assume that the greatest timestamps used before this operation occurred are 14.1, 15.2 and 16.3 respectively in hosts 1, 2 and 3.

In the first step, the incoming messages are received in a different order at each destination. All the messages are stored in a reception queue, timestamped and labelled as *undeliverable* (*ud*). In the second step, *host1* (the emitter of *m1*) collects the acknowledgements of *m1* with the proposed timestamps (16.1,17.2,17.3). After receiving all the timestamps, `host1` picks up 17.3, the greatest timestamp from this list, and multicasts the m1 final timestamp (17.3). The receivers label *m1* as `deliverable` (*DL*) and assign *m1* the final timestamp (17.3). Each host then reorders the waiting messages in ascending order of timestamps. Message *m1* will not be delivered until all messages preceding it in the reception queue have been delivered (as is the case of *m1* in `host3`). In the opposite case, *m1* remains waiting in the queue until it becomes head of the queue (i.e. *m1* in `host1` and `host2`). In steps *3* and *4* the same algorithm is executed on *m2* and *m3*.

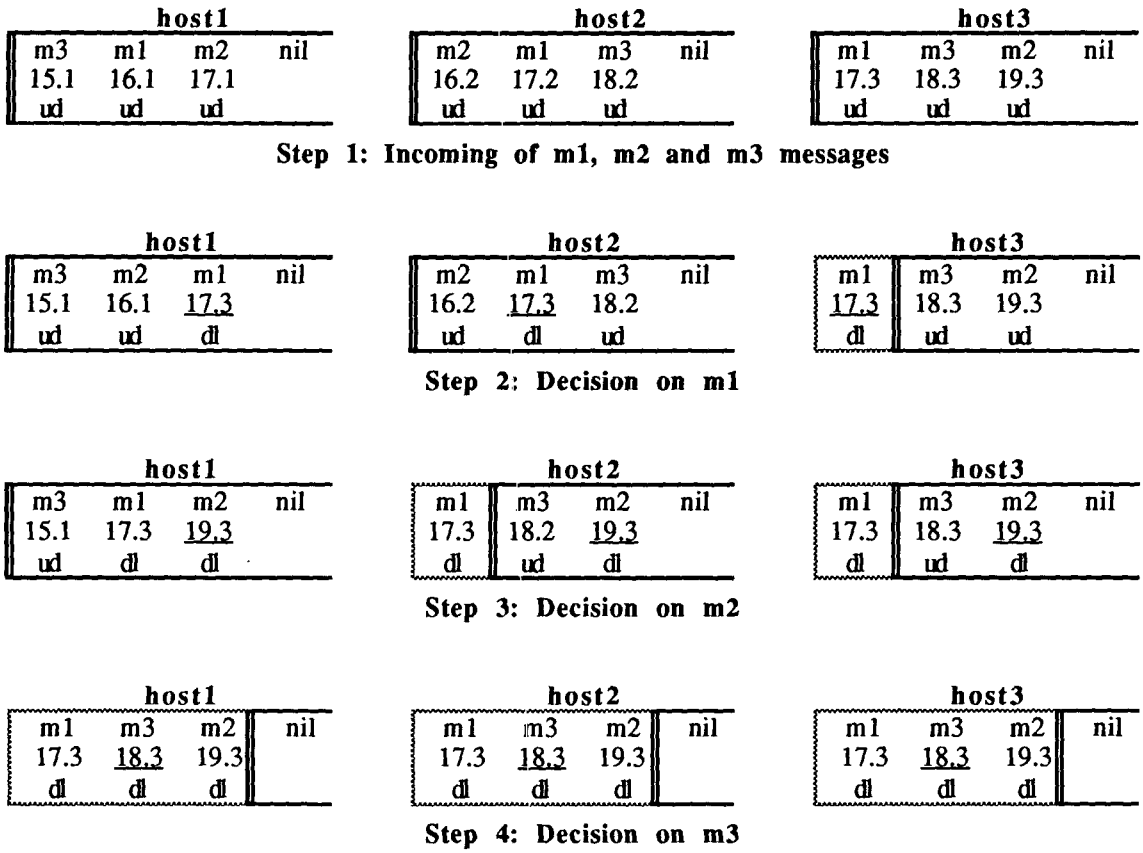

Figure 1

2.3. SOME REMARKS ON THE ABCAST PROTOCOL

We have defined the latency time as the interval of time expired between the moment a host receives a message and the moment the host effectively delivers it to the upper layer.

Several lessons can be drawn from the example described above:

(1) *Latency time* of received messages may differ (by orders of magnitude) on the same host.

(2) *Latency time* of the same message may differ (by orders of magnitude) on different hosts.

(3) *Latency time* may be divided into two intervals of time, *t1* and *t2* (shown in Figure 2).

- *t1* represents the expired time between the moment a host receives a new message and the moment it receives the final timestamp for this new message.
- *t2* represents the expired time between the moment a host receives the final timestamp of a message (the message is labelled deliverable) and the moment it effectively delivers the message to the upper layer.

(4) *t2* depends on the position of the message in the reception queue when the message is timestamped (labelled as deliverable) and how much time it takes to receive the final timestamps of all the previous messages in the queue.

(5) A deliverable message may wait unnecessarily for the timestamping of a message preceding it in the reception queue if the final timestamp of the preceding message is greater than the final timestamp of the waiting message. This is the case of *m1* in *host1*; it waited until *m3* was timestamped in the 4th step to be effectively delivered, even though *m3* received a final timestamp greater than that received by *m1*.

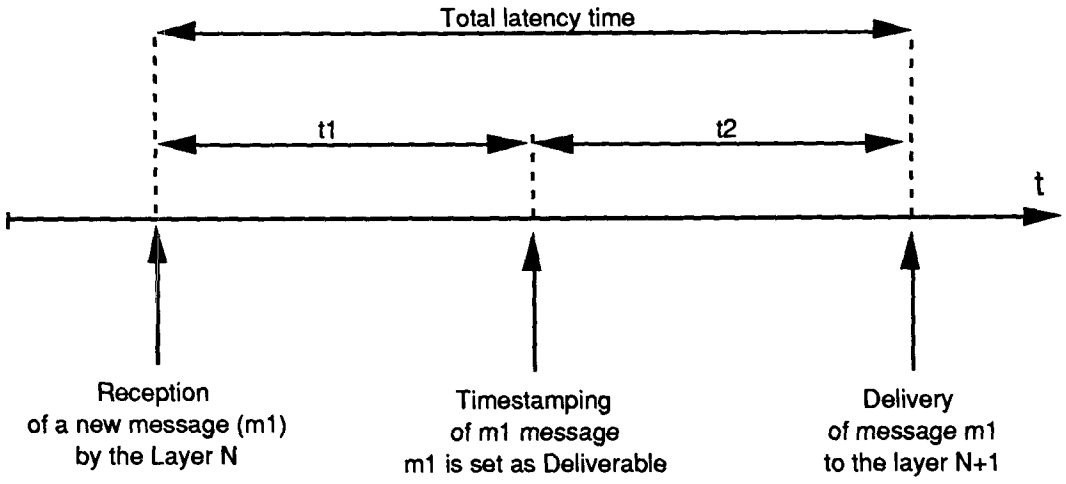

Figure 2

3. THE NEW ALGORITHM

3.1. DESCRIPTION

The example described above shows how a message may be timestamped and labelled as deliverable on a host but remain waiting in the reception queue only because one or more messages were received before it but were not yet timestamped. In all cases, a message must wait until all its predecessors are timestamped on a given host. This does not take into account the order in which these messages were received on the other hosts. This method may thus involve a long and unnecessary latency time, especially if the timestamp assigned to the preceding message of a undeliverable message is greater than that of the waiting (deliverable) message.

The solution proposed in this paper reduces the delay *t2*, the second component of the latency time defined above, and increases the probability of delivery of a message as soon as it is timestamped. In this algorithm, the *t2* delay of a particular message becomes less dependent on the *t1* delay of the other messages. This new solution is based on the same principles of the two-phase commit used in the *ABCAST* protocol, but uses a new mechanism for the total ordering by exchanging more information when acknowledging or timestamping messages to reduce the *t2* delay. Overall this solution allows the establishment of a partial order of messages in the waiting queues as the reception of the final timestamps progresses. This is done by exploiting information concerning the messages received by the other hosts and the way the message was timestamped. For example, it is unnecessary for a message to remain temporarily timestamped by a weak value on a given host whereas the same message has received a greater timestamp on another host; as soon as a host receives this type of information, it must update the timestamp value of the message in question and reorder its reception queue. The protocol runs as follows:

A host multicasts a message and waits for the acknowledgments of this message by the set of hosts belonging to the group. Each member acknowledges the received message. At the same time, and in addition to the timestamp assigned locally to the new message, the receiver host sends the state of its reception queue to the sending host including the order in which the waiting messages are stored, their identifiers and their assigned timestamps. To restrict the amount of information sent over the network, each member communicates only the next message identifier expected from each host and the timestamp it will assign to it. For example, if the reception queue of *hostA* contains 3 messages (*mb1,mb2,mb3*) waiting to be timestamped by *hostB* (the initiator host), *hostA* only sends the identifier of *mb3 + 1*. As soon as the sending host receives all the acknowledgements from all the hosts, it multicasts a *'validation message'* including the greatest value of the proposed timestamps for this message. At the same time, the sending host sorts the received information concerning the global states of all the reception queues of the different members. After sorting this information, the sending host proposes a partial order of the messages waiting to be timestamped. Thus, each host receives the final timestamp of the timestamped

message and at the same time it is able to situate the timestamped message in relation to the other waiting messages by exploiting the partial order proposed by the emitter.

3.2. PROTOCOL DESCRIPTION

3.2.1. INTRODUCTION AND ASSUMPTIONS

As noted above, the aim of this new protocol is the total ordering of the multicast message sent to a set of hosts belonging to a known and static group. Problems such as transmission errors, data flow control and re-transmission policy are assumed to be handled by a lower layer. Therefore, it is assumed that each multicast message is received by the set of hosts belonging to the group. The sequencing of messages received from the same source is assumed to be assured by the lower layer or by an external module not discussed here.

3.2.1.1. DEFINITIONS

ng:	The number of hosts belonging to the multicasting group.
mi:	A message multicast by the host *hi*.
mi_id:	Identifier of message *mi*.
local_ts:	The local timestamp variable incremented after each new message reception.
mi_ts:	The timestamp assigned to a message *mi*.
mi_state:	The state of *mi*. It indicates the state of a waiting message in the reception.queue. The state of a waiting message *mi* may be deliverable (*dl*) or undeliverable (*ud*).
next_mid:	The vector of the identifiers of the next expected message from each host at a given host.
next_mid(k):	Identifier of the next expected received message from the host *hk*.
queue_hi:	Reception queue of host *hi*.
ack_mi:	The acknowledgment message of the message *mi*.
ack_mi_ts:	The timestamp assigned locally to *mi* and included in *ack_mi*.
ackmi_next_mid:	The message Identifier vector included in the acknowledgment message *ack_mi*.
ackmi_next_mid(k):	Identifier of the next message expected from the host *hk*.
ts_mi:	The timestamp message multicast by the initiator host (*hi*) to validate *mi*.
tsmi_ts:	The final timestamp value assigned to *mi* and included in the timestamp message *ts_mi*.
max_next_ts:	This vector is included in the *ts_mi* message and described below.
min_next_mid:	This vector is included in the *ts_mi* message.

3.2.2. DISSEMINATION PHASE

3.2.2.1. MESSAGE EMISSION

Each message is identified by a unique sequence number called message identifier. The identifier is assumed to be strictly increasing and monotonic. A host *hi* prepares a message and multicasts it to a set of hosts belonging to a group .

3.2.2.2. MESSAGE RECEPTION

After receiving *mi*, a new message sent by host *hi*, the host *hj* executes the following actions:

- Increment the Local timestamp variable :

```
local_ts <-- local_ts + 1
```

- Assign to the message *mi* the local timestamp:

```
mi_ts    <-- local_ts
```

- Set the *mi* status as *undeliverable*:
  mi_status <-- undeliverable (ud).
- Update the identifier of the next expected message from host *hi*:
  next_mid(hi) <-- mi_id + 1
- Push *mi* in the reception Queue (*queue_hj*)
- Prepares the acknowledgement message *ack_mi* :

  ackmi_ts <-- local_ts (sets the timestamp of *mi in ack_mi*)

  Copy the *next_mid* vector to *ackmi_next_mid* vector
- Send the acknowledgement message *ack_mi*

3.2.2.3. ACKNOWLEDGEMENT RECEPTION

After receiving *ack_mi*, the acknowledgement message of *mi*, from a host *hj*, the source host executes the following actions :

- Update the list of hosts having acknowledged *mi*.
- Updates the *tsmi_ts* value:
  tsmi_ts <-- Maximum(*tsmi_ts, ackmi_ts*)
- Updates the *max_next_ts* and the *min_next_mid* vectors:
  For all *j* such that : *0 < j < ng*
  if (*ackmi_next_mid(j) < min_next_mid(j)*) then
  min_next_mid(j) <-- *ackmi_next_mid(j)* (1)
  max_next_ts(j) <-- *(ackmi_ts + 1)* (2)
  else if (*ackmi_next_mid(j) == min_next_mid(j)*) then
  max_next_ts(j) <-- Maximum (*max_next_ts(j)* , *ackmi_ts+1)*
- Wait for the acknowledgements from the remaining hosts

Note: (1) we store the lesser identifier of the message received from the host hj. (2) We store the lesser timestamp that might be assigned to the message identified by *ackmi_next_mid(j)*.

3.2.3. DECISION PHASE

3.2.3.1. DECISION

When all hosts have acknowledged *mi*, the source host (*hi*) will execute the following actions:

- Prepare *ts_mi* by including *max_next_ts*, *min_next_mid* and *tsmi_ts* into it.
- Multicast *ts_mi*.

3.2.3.2. TIMESTAMP RECEPTION

Each host receives the *ts_mi* timestamp message carrying *min_next_mid*, *max_next_ts* and *tsmi_ts* fields. *tsmi_ts* indicates the final timestamp of the waiting message *mi*. *min_next_mid*, *max_next_ts* help the receiving host to anticipate the decision on the waiting messages and guests their nearest timestamp values by the following way:

For each message (*mk*) from a host *hk* waiting for a timestamp at a host *hj*:

- If the identifier of *mk* (*mk_id*) is greater or equal than *min_next_mid(k)*, the final timestamp of *mk* (*mk_ts*) must be greater than or equal to *max_next_ts(k)*. Recall that *max_next_ts(k)* is less than or equal to *tsmi_ts + 1*. More precisely, *mk* will have a timestamp equal to the greatest value of: *mk_ts* and *(max_next_ts(k) + Δ)*; with *Δ = mk_id - min_next_mid(k)*. This is because *min_next_mid(k)* means that the lowest sequence number of the next expected message from the host *hk* was *min_next_mid(k)* at the moment *mi* was received by all the hosts. This implies that the message identified by *min_next_mid(k)* will have at least the value *max_next_ts(k)* as timestamp and a

message *mk* having an identifier greater than *min_next_mid(k)* will logically have a timestamp greater or equal to the maximum of *mk_ts* and *max_next_ts(k) + Δ*.

- Otherwise no anticipation could be done on *mk* if *mk_id* is lower than *min_next_mid(k)*

Thus *hj* host must do the following actions when it receives the timestamp message *ts_mi* :

- assign to *mi* the final timestamp:
  mi_ts <-- *tsmi_ts*
- declare the *mi* message as deliverable:
  mi_status <-- *deliverable (dl).*
- update the timestamp of the waiting messages in the reception queue:
  For all messages *mk* sent by *hk*:
  If *mk_state* == *undeliverable* then
  Δ <-- *mk_id - min_next_mid(k).*
  if *(Δ >= 0)* then
  mk_ts <-- Maximum (*mk_ts, max_next_ts(k) + Δ*)
- re-order waiting messages in the reception queue
- when the status of the first message in the reception queue is *deliverable*:
  Remove the first message from the reception queue and deliver it to the upper layer.

3.3. TOMP: ILLUSTRATION BY AN EXAMPLE

To show the benefits of this new algorithm and the way it enhances the ABCAST protocol we shall apply the protocol to the same example described in section 2.2.

Step 1: The *m1, m2, m3* messages are received at *host1, host2* and *host3* in a different order. Each host timestamps the received messages following the order it receives it. So *host1* assigns to *m1, m2* and *m3* the timestamps 16.1,17.1 and 15.1 respectively. *host2* assigns the timestamps 17.2,16.2, and 18.2 to the same messages even though *host3* assigns the values 17.3,19.3 and 18.3.

Step 2: In this step we assume that *m1* was the first message acknowledged by the set of hosts. Figure 3 depicts the *ackm1_next_mid* vectors submitted by each host when acknowledging *m1* and also the timestamps proposed for *m1*. So *host1* multicasts the vector *(m1_id+1, m2_id, m3_id+1)*. This vector indicates that after the reception of *m1*, *host1* expects *m1_id+1* as next message from *host1*, *m2_id* from *host2* and *m3_id+1* from *host3*. The timestamp proposed by *host1* for *m1* *ackm1_ts* is 16.1. Each host executes the same algorithm and proposes an *ackm1_next_mid* vector and an *ackm1_ts*. Finally *host1* receives the following *ackm1_next_mid* vectors: *(m1_id+1, m2_id, m3_id+1)*, *(m1_id+1, m2_id+1, m3_id)* and *(m1_id+1, m2_id, m3_id)* from *host1*, *host2* and *host3* respectively, and as *ackm1_ts* values: 16.1,17.2,17.3 from *host1*, *host2* and *host3* respectively. *host1* then determines the *min_next_mid* and *max_next_ts* vectors and *tsm1_ts* value. The value of *tsm1_ts* proposed is 17.3 which is the greatest value proposed by all hosts. The *min_next_mid* and *max_next_ts* calculated, following the algorithm presented above, are *(m1_id+1, m2_id, m3_id)* and (18.3,18.3,18.3).

Step 3: Each host receives *min_next_mid, max_next_ts* vectors and *tsm1_ts* included in the *ts_mi*. *m1_id+1* (resp *m2_id* and *, m3_id)* means that all the hosts had received at least the message *m1_id* (resp. *m2_id-1* and *m3_id-1)* and the next expected message from *host1* (resp. *host2* and *host3)* is *m1_id+1* (resp. *m2_id* and *m3_id*) and its timestamp will be at least 18.3 (resp 18.3 and 18.3) . Then, each host assigns to *m1* the proposed timestamp (17.3), declares it *deliverable*, and then checks the other waiting messages. For example, *host1* has *m3*(15.1), *m1*(16.1) and *m2*(17.1). According to *min_next_mid* and *max_next_ts*, *m3* (resp. *m2)* may be timestamped by a greatest value 18.3 (resp. 18.3), because *min_next_mid* indicates that all the hosts had received *m3_id-1* (resp. *m2_id-1)* and that they expect *m3* (resp. *m2)* as next message from *host3* (resp. *host2)* and its timestamp will be at least 18.3 (resp. 18.3). So since the local timestamp 15.1 (resp. 16.1) of *m3* (resp. *m2*) is lesser than 18.3 (resp. 18.3), *host1* will set *m3* (resp. *m2*) to the greatest value. The Figure 3, step3 depicts the state of the system once each host has executed the same algorithm after

receiving *ts_m1* and rearranged its reception queue. Notice that at the end of this step, *m1* may be delivered to the upper layer at the 3 hosts.

Afterwards, *m2* and *m3* follow the same processing in steps 4, 5, 6 and 7 (see also Figure 4).

Queue_R host1

m3	m1	m2	nil
15.1	16.1	17.1	
ud	ud	ud	

Queue_R host2

m2	m1	m3	nil
16.2	17.2	18.2	
ud	ud	ud	

Queue_R host3

m1	m3	m2	nil
17.3	18.3	19.3	
ud	ud	ud	

Step 1: Reception queue after reception of m1, m2 and m3

host1	**host2**	**hots3**	**ts_m1**	
ackm1_ts= 16.1	ackm1_ts= 17.2	ackm1_ts = 17.3	tsm1_ts = 17.3	
ackm1_next_mid	ackm1_next_mid	ackm1_next_mid	min_next_mid	max_next_ts
m1_id + 1	m1_id + 1	m1_id+1	m1_id + 1	18.3
m2_id	m2_id + 1	m2_id	m2_id	18.3
m3_id + 1	m3_id	m3_id	m3_id	18.3

Step 2:Acknowledgement + timestamping of m2

Queue_R host1

m1	m2	m3	nil
17.3	18.3	18.3	
dl	ud	ud	

Queue_R host2

m1	m2	m3	nil
17.3	18.3	18.3	
dl	ud	ud	

Queue_R host3

m1	m3	m2	nil
17.3	18.3	19.3	
dl	ud	ud	

Step 3: The reception queues after the decision on m1

host1	**host2**	**hots3**	**ts_m2**	
ackm2_ts = 17.1	ackm2_ts = 16.2	ackm2_ts = 19.3	tsm2_ts = 19.3	
ackm2_next_mid	ackm2_next_mid	ackm2_next_mid	min_next_mid	max_next_ts
m1_id + 1	m1_id	m1_id+1	m1_id	17.2
m2_id + 1	m2_id + 1	m2_id+1	m2_id+1	20.3
m3_id +1	m3_id	m3_id+1	m3_id	17.2

Step 4: Acknowledgement + timestamping of m2

Queue_R host1

m1	m3	m2	nil
17.3	18.3	19.3	
dl	ud	dl	

Queue_R host2

m1	m3	m2	nil
17.3	18.3	19.3	
dl	ud	dl	

Queue_R host3

m1	m3	m2	nil
17.3	18.3	19.3	
dl	ud	dl	

Step 5: The reception queues after the decision on m1 and m2

Figure 3

host1	host2	hots3	ts_m3	
ackm3_ts = 15.1	ackm3_ts = 18.2	ackm3_ts = 18.3	tsm3_ts = 18.3	
ackm3_next_ts	ackm3_next_ts	ackm3_next_ts	min_next_mid	max_next_mid
m1_id	m1_id +1	m1_id+1	m1_id	16.1
m2_id	m2_id + 1	m2_id	m2_id+1	19.3
m3_id + 1	m3_id+1	m3_id+1	m3_id	19.3

Step 6: Acknowledgement + timestamping of the m3 message

Queue_R host1			
m1	m3	m2	nil
17.3	18.3	19.3	
dl	dl	dl	

Queue_R host2			
m1	m3	m2	nil
17.3	18.3	19.3	
dl	dl	dl	

Queue_R host3			
m1	m3	m2	nil
17.3	18.3	19.3	
dl	dl	dl	

Step 7: The reception queues after the decision on m1, m2 and m3

Figure 4

4. CONCLUSION

In this paper we pointed out some weaknesses in the existing reliable multicast protocols and their origins. We discussed the problems caused by these weaknesses and proposed a solution to one of these problems, based on a protocol that enhances the latency time of the multicasted messages. It can be directly applied in the ISIS system by modifying the ABCAST protocol. However our solution could be used in any ordered multicast protocol which is based on the two-phase commit mechanism. We have simulated the two algorithms (the algorithm used in ABCAST and our new solution) on a SUN3/60 system. The simulation shows approximately 30-40% reduction of the latency time by the new solution in relation to the ABCAST protocol. We continue to test, analyse and investigate the modifications of the ABCAST protocol on ISIS system involving the algorithms discussed above.

5. ACKNOWLEDGMENTS

I am grateful to Patrick Pleinevaux and Guy Genilloud for their useful comments, suggestions and criticisms on the earlier draft of this article.

6. REFERENCES

[BirJos87] K. Birman and T.A Joseph , "Reliable communication in the presence of the failures", ACM Transactions On Computer Systems, Vol. 5, 1, Feb 1987.

[BirJos88] K. Birman and T.A Joseph , "Reliable broadcast protocols", Lecture Notes of Arctic 88, Tromso, July 1988.

[Birman88] K. Birman et al. "ISIS - A Distributed Programming Environment", Cornel University, June 1988.

[Chang84] Chang J.M and Maxemchuck, "Reliable broadcast protocols", ACM Transactions On Computer Systems Vol. 2, 3, pp251-273, Aug 1984

[Gray 78] Gray J. "Notes on database operation systems". Lecture Notes in Computer Science, 1978.

[Lamport78] L. Lamport, Time, clocks, and the ordering of events in a distributed system" ACM Communications Vol. 21, 7, July 1978.

[Schiper89] A. Schiper, J. Eggli, A. Sandoz, "A new algorithm to implement causal ordering" Proc. of the 3th International Workshop on Distributed Algorithms, Nice, Sept 1989.