Implementation and Performance of a Total Order Multicast to Multiple Groups

Udo Fritzke Jr.*

Pontifícia Universidade Católica de Minas Gerais

Campus de Poços de Caldas

37701-355 Poços de Caldas-MG, Brazil

E-mail :udo@pucpcaldas.br

Abstract

Group communication proved to be a powerful tool for the construction of reliable distributed applications. This paper presents an implementation and the performance of a total order (or atomic) multicast protocol. This communication primitive assumes the distributed system is composed by a set of disjoint process groups and allows messages to be multicast to arbitrary sets of groups. The protocol implements total ordered delivery of messages. Moreover, message deliveries are reliable, that is, there is no message duplication nor spurious messages, and messages are delivered to all correct recipients or none of them.

We outline the total order protocol and then we present a prototype architecture. The prototype implementation allowed us to obtain performance figures that stress the role of underlying agreement services, namely a consensus primitive. We also compare analytically the costs of the protocol we implemented with the costs of other protocols that ensure the same delivery properties.

1 Introduction

Many distributed applications that execute on asynchronous networks subject to failures, need both consistent sharing of data and fault tolerance. Group communication primitives, in particular multicast, have been used in several applications to hide fault-tolerance and asynchrony issues from application programmers. As for example, recent

*This work was supported by a grant of the CNPq/Brazil.

research show how transaction systems on replicated data can benefit from reliable and total ordered multicast in order to verify important properties, as data consistency, deadlock avoidance, reduction of the number of aborted transactions and implementation of non-blocking distributed atomic commitment (e.g. [17, 8]).

Multicast with order and reliability properties are however based on complex underlying services. The implementation of reliable end-to-end message transfer and agreement on the message delivery order among destination processes are far of being trivially implemented in the presence of message losses, faults of processes and communication links, and arbitrary processing and message transfer delays. Pragmatically, reliability and order can be treated independently. Reliable message transfer imposing low overhead to scalable raw multicast primitives (like IP-multicast) is a subject of intense research (e.g. [6, 14, 1]). Ordered message delivery has also been studied in a range varying from weaker order constraints, as causal order (e.g [2, 4]), to stronger ones, as global total order [9, 16, 11].

In this paper we focus on the implementation of a *total order* (or *atomic*) *multicast to multiple groups* primitive. The target of the multicast is a set of groups dynamically defined by a parameter of the multicast. The total order multicast enforces reliability and global total order in message deliveries. Reliability ensures that there is no message duplication nor spurious messages, and messages are delivered to all correct recipients or none of them (if the sender crashes during the multicast and no process receives the message). Global total order respects an acyclic relation $<$ defined on messages: if a process delivers m before m' we have $m < m'$.

We review in this paper the principles of a protocol proposed in [9] implementing total order multicast. That protocol delivers messages according to timestamps that are associated with messages by recipient processes. The coherence of timestamps computed by distinct processes is reached thanks to the use of a Consensus primitive [5].

The prototype we have developed intended to study the behavior of this protocol, particularly, when it comes to the agreement on the order of message deliveries. The experimental results obtained highlight the role of the consensus primitive for the performance of total order multicast. We show here that in order to obtain acceptable performances we need a consensus primitive able to treat sets of messages.

This paper is structured as follows. Section 2 defines the distributed environment, multicasts and consensus. Section 3 outlines the protocol and the implementation of the total order multicast primitives. This section also compares analytically its costs with the complexity of related work. Section 4 details experimental results on the behavior of the total order protocol. Section 5 concludes the paper.

2 Group communication model

This section presents definitions and assumptions on the distributed environment as well as the definitions of the communication primitives.

2.1 Process groups, failures and asynchronous systems

We consider a distributed system composed of a finite set Π of processes $p_1, p_2, \ldots, p_n$. The set Π of processes is statically structured into non-empty non-intersecting groups g_x, $g_y, \ldots, g_z$ (*i.e.*, $\forall x : g_x \neq \emptyset \wedge \cup_x g_x = \Pi \wedge \forall x, y, x \neq y : g_x \cap g_y = \emptyset$).

A process can fail by *crashing*, *i.e.*, by prematurely halting. A process behaves according to its specification until it possibly crashes. By definition a *correct* process is a process that never crashes. A crashed process does not recover from the failure. We consider that groups are *reliable*, that is, a majority of processes are correct in each group (let f_x be the maximum number of processes of the group g_x that can crash, then $\forall x : f_x < |g_x|/2$).

Every pair p_i and p_j of processes is connected by a reliable channel, ensuring that every message sent by p_i to p_j, is eventually received by p_j if p_i and p_j are correct. We do not assume any bound on process relative speed and on message transfer delays, that is, we consider that the distributed system is *asynchronous*.

2.2 Multicast to multiple groups

We consider two multicast specifications: *reliable multicast* and *total order multicast* (or *atomic multicast*). Reliable multicast is defined by two primitives, namely, R_multicast(m) and R_deliver(m). R_multicast(m) allows a process to send a message m to the processes of a set of destination groups, designated by $m.dest$. R_deliver(m) allows a process to deliver the message m sent by the invocation of R_multicast(m). Total order multicast is also implemented by two primitives, TO_multicast(m) and TO_deliver(m), that correspond, respectively, to R_multicast(m) and R_deliver(m). When a process executes TO_multicast(m) (or R_multicast(m)) we say that it "TO-multicasts" m (or "R-multicasts" m). When a process executes TO_deliver(m) (or R_deliver(m)) we say that it "TO-delivers" m (or "R-delivers" m).

The semantics of reliable multicast primitives is defined by the following three properties[1] [11]:

- *Uniform Validity. If a process p R-delivers m, then some process has R-multicast m and p belongs to a group g such that $g \in m.dest$.* This property expresses there are no spurious messages.

[1] The process that is mentioned in an *uniform* property can crash later, but until it (possibly) crashes it is *a priori* considered as being *potentially* correct. Conversely, a non-uniform property concerns only correct processes.

- *Uniform Integrity. A process R-delivers a message m at most once.* This property expresses that there is no message duplication.

- *Termination. If (1) a correct process R-multicasts m, or if (2) a process R-delivers m, then all correct processes that belong to a group of $m.dest$ R-deliver m.* This property defines that the only case in which a multicast can not terminate is when the sender process crashes during its invocation of R_multicast(m), and none of the destinations has delivered the message.

The semantics of total order multicast is defined by the before mentioned *Uniform Validity*, *Uniform Integrity*, and *Termination* properties, additioned to the following order property:

- *Global Total Order. Let "$<$" be the relation on messages defined in the following way: if a process TO-delivers m_1 before m_2, then $m_1 < m_2$. The relation "$<$" is acyclic.* This property expresses that the set of delivered messages can be totally ordered (by doing a topological sort of "$<$") in a way consistent with the message delivery order at each process.

2.3 Consensus

Informally, in the Consensus problem a set of processes can propose (possibly distinct) values, and subsequently, decide on some value v that is some of the proposed values. Moreover, all correct processes eventually decide some value, and no two processes decide on different values. The proposition of a value v to a consensus execution is done by the primitive Propose(v), and a v consensus decision value is obtained calling Decide(v).

It has been demonstrated by Fischer, Lynch and Paterson [5] that the Consensus problem has no deterministic solution in asynchronous distributed systems that are subject to even one process crash failure. This impossibility result is related to the non accurate distinction between crashed and slow processes in asynchronous systems. However, if each process of a group can be provided with sufficiently complete and accurate process failure information, one can build a protocol solving Consensus on such systems. Chandra and Toueg showed in [3] that Consensus is solved if: (1) every correct process in the system is able to eventually (and permanently) suspect every crashed process (a liveness property), and also if (2) after some time, a correct process is never suspected by any process (a safety property).

In the notation introduced in [3], these properties are held by "unreliable" failure detectors of class $\diamond S$ (or "Eventually Strong" failure detectors). So, in order to be able to use a consensus protocol, we assume that each process is provided with a failure detector of class $\diamond S$.

3 Implementation of a Total Order Multicast

This section describes an implementation of the TO_multicast(m) and TO_deliver(m) primitives, namely, the protocol proposed in [9]. Hereafter, this protocol will be cited within the text as TOM (for "Total Order Multicast"). The TOM protocol is based on three underlying services: a primitive solving the Consensus problem [5], a reliable multicast implementing the properties defined in subsection 2.2, and unreliable failure detectors of class $\diamond S$ [3].

3.1 Overview of the protocol

We outline here the structure of the TOM protocol. Protocol details and a correctness proof of it can be found in [9].

The global total order property of the TOM protocol is provided by the association of a timestamp with each message and by delivering messages according to the order defined by their timestamps. A timestamp is a pair (clock value, group identity). The timestamp associated with a message m is denoted $m.ts$. The clock values of timestamps are values of a *group clock* that equips each process, and that is updated both upon the reception of a TO-multicast message and when some message gets its (definitive) timestamp. The group clock of a process p_i is implemented by a variable $clock_i$, initially set to 0.

A message m is TO-multicast to groups of servers by the means of an invocation of the R_multicast(m) primitive, that implements the reliable delivery of m to the destination groups. Thanks to the reliability assumption on the communication channels, this can be done by the following straightforward protocol:

- R_multicast(m) is executed by sending m to all processes in groups of $m.dest$.
- When a process receives m the first time, it forwards it to all other processes in groups $g_x \in m.dest$, and only then R-delivers m. Otherwise, m is ignored.

Now, let m be a message that has been TO-multicast to a set of groups, say g_x and g_y, that is $m.dest = \{g_x, g_y\}$. Suppose m was R-delivered and then stored in the reception queue of a process $p_i \in g_x$ of some group $g_x \in m.dest$. This starts the computation of the TO-delivery of m at p_i. In order to obtain a global total ordered delivery, the protocol proceeds in four consecutive steps.

The figure 1 illustrates the execution steps. Arrows represent the flow of time at a sender process and recipient processes, as well as the messages that are exchanged. Squares represent an execution of a consensus protocol.

Step 1 (Timestamping consensus). Each group $g_x \in m.dest$ defines a timestamp for m (denoted $m.ts^x$). This timestamp is g_x's proposal to be the definitive timestamp for m. To obtain a coherent timestamp proposal, in the scope of group g_x, it is used

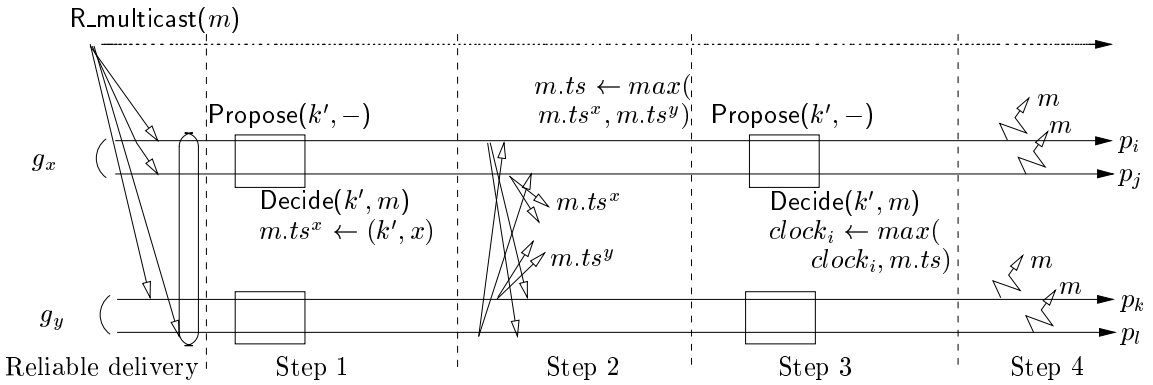

Figure 1: The execution of a total ordered multicast of a message m to two groups of processes.

a k-numbered execution of a consensus[2]. In a process $p_i \in g_x$, the group clock $clock_i$ is incremented by one and then k assumes its present value. A consensus execution is started by a process every time some message is R-delivered to the process and stored in the local reception queue. The oldest message in the queue, say m', is the proposed value for that k^{th} execution. When the message m becomes the decision of some k'^{th} consensus, the number k' becomes the agreed clock value of m in group g_x ($m.ts^x$).

Step 2. Each group of $m.dest$ proposes its timestamp for m to the other groups of $m.dest$. This is done by sending $m.ts^x$ to all processes of groups in $m.dest$. Then each group computes the greatest timestamp proposed for m. Let $m.ts$ be this greatest timestamp: it is the definitive timestamp associated with m (by construction, it is the same for all groups).

Step 3 (Resynchronization consensus). After the definition of m's definitive timestamp, the group clock has to be updated with a value greater than or equal to the clock value of $m.ts$ to ensure progression of group clock and the delivery of messages, even if there is no more messages exchanged. Moreover, the group clock has to be updated by any process within the group in a consistent manner. To accomplish this, for each message m' that has a definitive timestamp, it is executed a second k-numbered consensus, k also being the present value of the group clock and m' being the proposed value. When a message with a definitive timestamp, say our message m, becomes the decision value of a k'^{th} consensus execution, the value of the group clock is set with the greatest value among the present group clock and the clock value of $m.ts$.

[2]As suggested in [3], a k-numbered consensus execution is easily obtained by adding a parameter k (that assumes a counter value) to the consensus primitives, and then tagging the consensus messages with k.

Step 4. Finally, the process throws m from the reception queue and TO-delivers it when $m.ts$ is the lowest timestamp among all the timestamped messages in the queue (be these timestamps proposals or definitive timestamps).

We note that two sequential consensus executions are then necessary for the delivery of a message, a first timestamping consensus and then a group clock resynchronization consensus. As we will see in section 4, this represents a performance limitation that can however be circumvented when a higher message throughput is expected from the protocol.

3.2 Related work

There are other protocols implementing TO_multicast(m) and TO_deliver(m) primitives in asynchronous systems. Rodrigues *et al.* propose a SCALATOM (for "SCALable ATOMic multicast) protocol in [16]. It also associates messages with timestamps, that are however calculated differently. Processes exchange timestamps proposals and define a final timestamp with a consensus execution that involves all processes belonging to destination groups of the message. Before the total order delivery of the message, the history of processes are also exchanged in the scope of each group. This will allow a process to safely deliver a message respecting global total order.

We can also use an atomic broadcast primitive, as the one in [3] to implement TO_multicast(m) and TO_deliver(m). An atomic broadcast also ensures reliable and total ordered delivery of messages, but messages are always addressed to all processes that compose the distributed system. By considering that the processes of the distributed system compose a unique group of processes, the atomic broadcast allows to build a global total order over this group. A total order multicast is than obtained by the atomic broadcast of any message to that fictitious group, and by delivering messages exclusively to processes belonging to actual destination groups. In fact, this corresponds to a "non minimal" protocol, that does not respect the Minimality property defined in [10].

3.3 Complexity comparisons

We compare in the following the complexity in number of messages and in number of communication steps of the TOM, the SCALATOM and the non-minimal protocols.

Let G be the number of groups in the distributed system. Let d be the number of destination groups of a message that is TO-multicast ($1 < d \leq G$), and n be number of processes of any group. Consider protocol executions without failure nor failure suspicion. The sender does not belong to any destination group.

The complexity of the TOM protocol is the overall complexity of a reliable multicast plus the complexity of the three initial steps described above (the fourth step does not

require communication).

Complexity of a reliable multicast. The simple protocol sketched in subsection 3.1 costs $d^2.n^2$ messages and one communication step.

Complexity of step 1. It is the cost of a consensus protocol execution. Several consensus protocols have been proposed in the literature. We consider two protocols, the protocol due to Chandra and Toueg [3], denoted here CT, and the one from Hurfin and Raynal [13], denoted HR. The CT protocol spends $n(n-1)+3(n-1)$ messages and four communication steps. HR costs $2n(n-1)$ messages and two communication steps.

Complexity of step 2. It is the cost of the exchange of timestamps proposals between processes in destination groups of m: $d(d-1)n^2$ messages and one communication step.

Complexity of step 3. It is the cost of a consensus protocol, and thus, the same complexity as step 1.

The first line in table 1 summarizes the total complexity of TOM, with the CT and HR consensus protocols. The complexities of both the SCALATOM and the non minimal protocols are shown in the subsequent lines of table 1. We note that both SCALATOM and the non minimal protocols also rely on reliable multicast and consensus primitives. A detailed deduction of the complexity of the consensus protocols and of the total order multicast protocols mentioned here can be found in [7].

Protocol	CT consensus		HR consensus	
	Number of messages	Number of comm. steps	Number of messages	Number of comm. steps
TOM	$2d^2.n^2 + d.n^2 + 4d.n - 6d$	10	$2d^2.n^2 + 3d.n^2 - 4d.n$	6
SCALATOM	$3d^2.n^2 + d.n^2 - 3$	7	$4d^2.n^2 + d.n^2 - 4d.n$	5
Non minimal	$2.G^2.n^2 + 2.G.n - 3$	5	$3.G^2.n^2 - 2.G.n$	3

Table 1: Comparisons between TOM, SCALATOM and the non minimal protocols, in terms of the number of messages and communication steps, considering the use of CT and HR consensus algorithms.

In opposition to the non minimal atomic multicast implementation, both TOM and SCALATOM are genuine atomic multicast protocols [10]. Accordingly, the complexities of TOM and SCALATOM are of $O(d^2)$, whereas the non minimal protocol is $O(G^2)$. When the distributed system is structured into non-intersecting groups and messages frequently only address some of them, genuine implementations become more efficient than non minimal ones. If we assume $G = 10$ we note that:

- When $d < 10$, TOM with the CT consensus algorithm requires less messages than the non minimal protocol.
- When the HR consensus is used TOM always outperforms the non minimal protocol.
- When $d \leq 8$ SCALATOM is better than the non minimal protocol, no matter the CT or the HR consensus is used.

The $O(d^2)$ complexity of TOM is due to the reliable multicast and to the exchange of timestamps proposals. For SCALATOM, the $O(d^2)$ complexity is due to the reliable multicast, to the exchange of timestamps proposals, and to the consensus protocol. The $O(d^2)$ cost of consensus in SCALATOM, in contrast to $O(n^2)$ consensus of TOM, makes SCALATOM more complex than TOM for all messages addressed to more than one group. In fact, both protocols differ in that only TOM respects a Locality property [9] that limits consensus executions to one process group.

With respect to the number of communication steps, the TOM algorithm reveals a worst behavior than SCALATOM. This is clearly due to the two consensus executions per message required by the former protocol. But this measure hides the fact that consensus executions in SCALATOM may expend more messages than in TOM when larger numbers of groups are addressed. This is particularly true when, additionally, multicast is build on top of point-to-point communication services.

3.4 The prototype

We have implemented a prototype of a group communication service based on the TOM algorithm outlined in subsection 3.1. The prototype also includes the required underlying services, providing a C++ library with the following components:

- A reliable communication service that implements group configuration primitives, the R_multicast(m) and R_deliver(m) primitives, and a reliable send primitive. This service was built on top of a stream socket layer (TCP).
- A failure detection service that provides each process with a list of processes suspected to be crashed. Failure detection is implemented with hints of the socket layer about interrupted TCP connections [15].
- A consensus primitive based on the HR [13] protocol. The HR protocol was adopted due to its lower (or at worst equal) cost in communication steps, when compared to other contemporary consensus protocols.
- The TO_multicast(m) and TO_deliver(m) primitives.

This implementation intended mainly to be a basis for experimentations with the TOM protocol. As it will be shown in the next section, we were interested in the analysis of the original protocol, sketched in this paper, as well as in the performances obtained after some optimizations on the use of consensus. To obtain an absolute performance of TOM, other optimizations should be envisaged. The use of multi-point communications instead of point-to-point TCP is one of them. To render unto raw multicast services (IP-multicast) the reliability features required by TOM would however have demanded a considerable additional implementation effort. Therefore, we decided to implement reliable communication channels with the help of TCP connections. This made easier the construction both of the reliable multicast primitives and of the failure detection mechanism. We note however that more efficient reliable multicast protocols have been studied and implemented elsewhere (e.g. [6, 14, 1]).

4 Performance of the protocol

The performance figures presented in this section were obtained by running the implemented TOM protocol on a distributed environment composed of nine Sun workstations (Sparc-5, Ultra-1 and Ultra-30) under the Solaris operating system, interconnected by a 10/100 Mbps Ethernet network. This measures allowed us to identify the bottlenecks in the protocols used in our communication architecture. We run a large set of experiments and we will discuss here some important results related to the TOM protocol, in failure free scenarios as well as in the presence of process crashes.

4.1 Failure free executions

Figure 2 shows the response of the protocol in terms of the number of TO-delivered messages per second, for different TO_multicast(m) invocation rates, in protocol executions without process failures. This delivery time is computed after a stream of 10,000 TO_multicast(m) invocations. Each TO_multicast(m) invocation addresses 3 process groups, each of them composed by 3 processes. A given invocation rate is obtained by defining a delay time between any two consecutive invocations of TO_multicast(m). Messages have lengths of 10 bytes, 500 bytes or 1,000 bytes.

The three curves in the lower part of figure 2 illustrate the behavior of the TOM protocol sketched in subsection 3.1. As it was described there, TOM requires two consensus executions before a message is allowed to be TO-delivered, the timestamping consensus and the group clock resynchronization consensus. These curves highlight the rapid saturation of the message delivery rate as the TO_multicast(m) invocation rate increases. This happens as soon as the time delay between two TO_multicast(m) invocations is smaller than the time required to TO-deliver a message. Even if there are several mes-

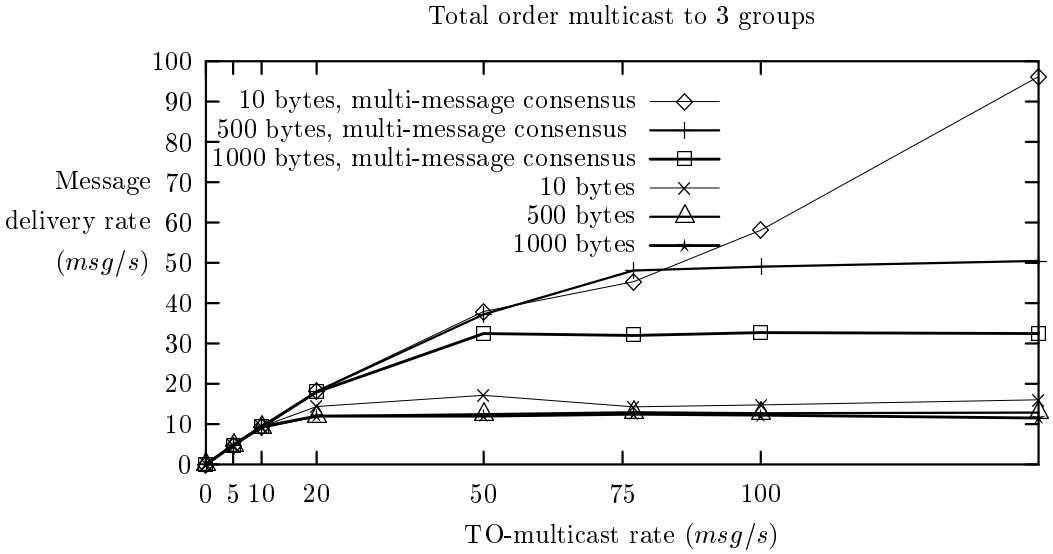

Figure 2: Total ordered message delivery rate (msg/s) as a function of the TO_multicast(m) invocation rate (msg/s).

sages in the local reception queue waiting to be treated by the TOM protocol, each of them is proposed in two sequential consensus executions. Consequently, the consensus executions acts in this case as a bottleneck before the final delivery of messages.

In order reduce this problem, consensus executions should propose and decide sets of messages, rather than only one message. Accordingly, when a consensus decides on a set of messages, all messages not yet timestamped get simultaneously their timestamps. Moreover, all decided messages already associated with a timestamp can contribute to the resynchronization of the group clock. With this modification, TOM is able, in some executions, to timestamp several messages and correctly re-synchronize group clocks after only one consensus execution. The modified protocol is detailed in [9, 7].

The three curves in the higher part of the figure 2 show the gains in terms of the message delivery rates provided by the optimization of the consensus use. The maximum message throughput of the protocol increased 2.7 to 5.6 times, with longer (1,000 bytes) and shorter (10 bytes) messages respectively. With longer messages, the saturation of the message delivery rate occurs after 32 messages per second invocation rate, whereas without a multi-message consensus, this happens around 12 messages per second. With smaller messages, no saturation at all occurred before the maximum emission rate that could be produced with our implementation (by setting the delay time between TO_multicast(m) invocations to zero). In fact, as the rate messages are R-delivered to the TOM protocol increases, the consensus protocol tends to decide on a greater set of messages. Accordingly, the number of consensus executions in the overall run decreases, and TOM manages better to respond to the TO_multicast(m) rate.

Table 2 presents the mean times for the delivery of messages with multi-message

consensus when 1, 2 and 3 groups, with 3 processes each, are addressed by messages with different lengths. This table was obtained by experiments on top of Ultra-1 and Ultra-30 Sun workstations.

Message length	1 group	2 groups	3 groups
10 bytes	2.4 ms	4.5 ms	5.2 ms
500 bytes	2.4 ms	8.6 ms	10.4 ms
1,000 bytes	3.0 ms	13.9 ms	16.0 ms

Table 2: Mean message delivery times (in milliseconds) of the TOM protocol.

4.2 Executions with failures

To verify the impact of failures in the TOM protocol, we also measured the message delivery rate after introducing process crashes within the protocol execution.

The HR consensus protocol we adopted in our prototype is based on a rotating coordinator scheme [3]. The actual coordinator process is responsible for the determination of a consistent decision value. For this, a consensus round is executed and, when no failure suspicion occur, a decision is obtained in this round. When the coordinator is suspected by other processes to be crashed, these processes chose deterministically[3] another coordinator and then they start another consensus computation round. As new coordinator processes are suspected to be crashed, successive rounds are executed before some decision is taken. Consequently, the duration of a consensus execution increases with failures of coordinator processes.

Figure 3 compares the message delivery rate of the TOM protocol without process crashes and considering one process crash per group. The curve with no failures corresponds to the previously discussed scenario (figure 2), where some process TO-multicasts 500 bytes long messages to 3 process groups. In a second curve, we forced a group coordinator crash just before any TO_multicast(m) invocation. This causes at least one more execution round within each consensus execution.

The curves in figure 3 reveal an interesting behavior of TOM during such a failure scenario. As any consensus takes more time to decide a value in the presence of a coordinator failure, more R-delivered messages are accumulated in the reception queues, before they become the proposed value of a next consensus. Thus, the consensus protocol tends to propose and decide on greater sets of messages and less consensus executions are required to the delivery of the TO-multicast messages. This also helps TOM to follow

[3]The choice of the new coordinator is based on globally well known values, as the number of processes in the group and the process id's.

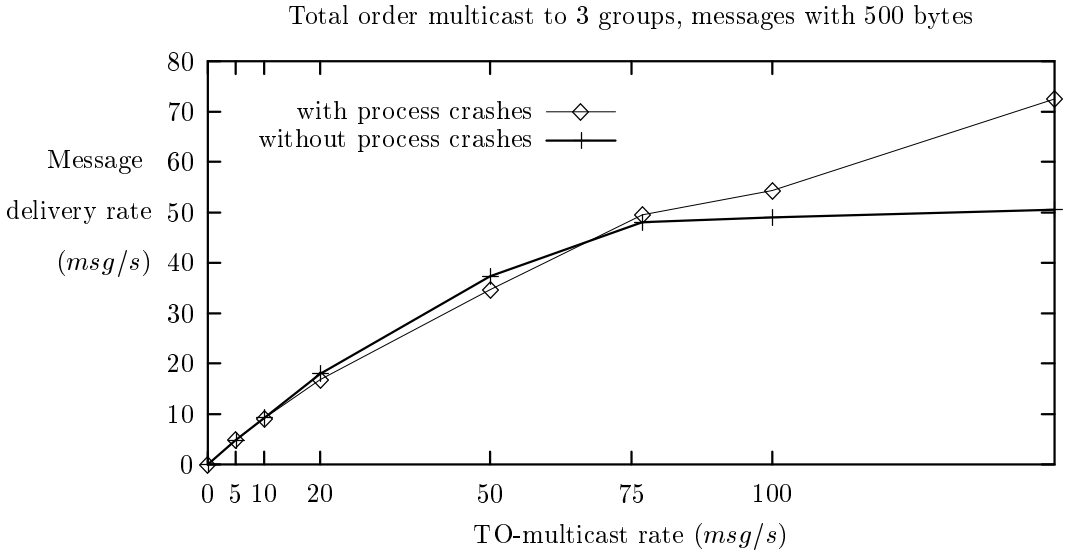

Figure 3: Total ordered message delivery rate (msg/s) as a function of the TO_multicast(m) invocation rate (msg/s), considering runs without process crashes and with one process crash per group.

higher messages throughput, what is represented in figure 3 by a smoother saturation of the delivery rate.

5 Concluding remarks

We presented here a prototype implementing a total order multicast primitive. Our implementation aimed mainly the study of the costs of the underlying consensus primitive. The use of consensus protocols able to handle sets of messages as proposed and decided values, allowed us to enhance the average message throughput up to 5.6 times, in relation to the initial protocol, based on single message consensus protocol. Additionally, we noted that failures do not necessary represent a bottleneck to the delivery of messages. The accumulation of messages in the protocol during higher message multicast rates, can reduce the number of consensus executions, and in some cases, enhance message delivery rates.

Our prototype does not present an optimal performance, that could be envisaged by using better reliable multicast protocols. Protocols that add reliability to raw multicast primitives should be useful in this case. However, the results give interesting perspectives with respect to applications of our total order multicast.

Recent research show how transaction systems on replicated data can benefit from reliable and total ordered multicast in terms of safety properties, as one-copy behavior of replicas, as well as liveness properties, as deadlock avoidance, reduction of the number of aborted transactions and simplification of non-blocking distributed atomic commitment

(e.g. [8, 12]). It is important to note that these problems are very difficult (when not impossible) to solve in asynchronous distributed systems. Total ordered multicasts for asynchronous systems hide a great part of this complexity from application programmers.

References

[1] Marinho Barcellos, André Detsch, Guilherme B. Bedin, and Hisham H. Muhammad. Efficient TCP-like multicast support for group communication systems. In *Proceedings of the IX Brazilian Symposium on Fault-Tolerant Computing (SCTF01)*, Florianópolis-SC, Brazil, March 2001.

[2] Kenneth Birman, André Schiper, and Pat Stephenson. Ligthweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[3] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(1):225–267, March 1996. (a preliminary version appeared in Proc. of the 10th ACM Symposium on Principles of Distributed Computing, pp. 325-340, 1991).

[4] P. Ezhilchelvan, R. Macêdo, and S. K. Shrivastava. Newtop: a fault tolerant group communication protocol. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 296–306, Vancouver, May 1995.

[5] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[6] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *IEEE/ACM Transactions on Networking*, volume 5, pages 784–803, December 1997.

[7] Udo Fritzke Jr. *Les systèmes transactionnels répartis pour données dupliquées fondés sur la communication de groupes.* PhD thesis, Université de Rennes 1, January 2001.

[8] Udo Fritzke Jr. and Philippe Ingels. Transactions on partially replicated data based on reliable and atomic multicasts. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS-21)*, Phoenix, Arizona, USA, April 2001.

[9] Udo Fritzke Jr., Philippe Ingels, Achour Mostefaoui, and Michel Raynal. Consensus-based fault-tolerant total order multicast. *IEEE Transactions on Parallel and Distributed Systems*, 12(2), February 2001.

[10] Rachid Guerraoui and André Schiper. Genuine atomic multicast. In *Proceedings of the International Workshop on Distributed Algorithms (WDAG'97), LNCS 1320*, pages 141–154, Saarbrücken, Germany, September 1997.

[11] Vassos Hadzilacos and Sam Toueg. *Distributed Systems, Second Edition*, chapter 5: Fault-Tolerant Broadcasts and Related Problems, pages 97–145. ACM Press New York and Addison-Wesley, 1993.

[12] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. Using multicast to reduce deadlock in replicated databases. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, Nurnberg, Germany, October 2000. IEEE.

[13] Michel Hurfin and Michel Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12:209–223, 1999.

[14] S. K. Kasera, G. Hjálmtýsson, D. F. Towsley, and J. F. Kurose. Scalable reliable multicast using multiple multicast channels. In *IEEE/ACM Transactions on Networking*, volume 8, pages 294–310, June 2000.

[15] Nuno Neves and W. Kent Fuchs. Fault detection using hints from the socket layer. In *Proceedigns of the 16th Symposium on Reliable Distributed Systems (SRDS'97)*. IEEE Computer Society, October 1997. Durham, North Carolina.

[16] Luís Rodrigues, Rachid Guerraoui, and André Schiper. Scalable atomic multicast. Technical Report DI-FCUL TR-98-2, Departamento de Informática, Faculdade de Ciências de Lisboa, Lisboa, Portugal, January 1998.

[17] I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 148–155, Amsterdam, The Netherlands, May 1998.