Fault-Tolerant Total Order Multicast to Asynchronous Groups

Udo FRITZKE Jr.* † Philippe INGELS Achour MOSTFAOUI Michel RAYNAL
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{ufritzke,ingels,achour,raynal}@irisa.fr

Abstract

While Total Order Broadcast (or Atomic Broadcast) primitives have received a lot of attention, this paper concentrates on Total Order Multicast to Multiple Groups in the context of asynchronous distributed systems in which processes may suffer crash failures. “Multicast to Multiple Groups” means that each message is sent to a subset of the process groups composing the system, distinct messages possibly having distinct destination groups. “Total Order” means that all message deliveries must be totally ordered.

This paper proposes a protocol for such a multicast primitive. This protocol is based on two underlying building blocks, namely, Uniform Reliable Multicast and Uniform Consensus. Its design characteristics lie in the two following properties. The first one is a Minimality property, more precisely, only the sender of a message and processes of its destination groups have to participate in the multicast of the message. The second property is a Locality property: no execution of a consensus has to involve processes belonging to distinct groups (i.e., consensus are executed on a “per group” basis). This Locality property is particularly useful when one is interested in using the Total Order Multicast primitive in large scale distributed systems. An improvement that reduces the cost of the protocol is also suggested.

1 Introduction

Total Order Broadcast (also called Atomic Broadcast) is one of the most important agreement problems encountered in the design and the implementation of fault-tolerant distributed systems. This problem consists in providing processes with a communication primitive that allows them to broadcast and deliver messages in such a way that processes agree not only on the set of messages they deliver but also on the order of message deliveries. Total order broadcast has been identified as a basic communication primitive in many systems (such as the ones described in [16]). It is particularly useful to implement fault-tolerant services by using software-based replication. By employing this primitive to disseminate updates, all correct (i.e., non-crashed) copies of a service are delivered the same set of updates in the same order, and consequently the state of the service is kept consistent.

It has been shown in [4] that Total Order Broadcast and Consensus are equivalent problems in asynchronous systems prone to process crash failures. So, all results attached to the consensus problem also hold for total order broadcast. More specifically, the Fisher-Lynch-Paterson’s impossibility result [9] also applies to total order broadcast. Namely, it is impossible to solve the total order broadcast problem in an asynchronous system prone to even a single process crash. To circumvent this impossibility result, Chandra and Toueg have introduced the Unreliable Failure Detector concept. More precisely, they have shown that it is possible to solve the consensus problem (and hence the total order broadcast problem) in an asynchronous distributed system equipped with unreliable failure detectors if those satisfy some minimal properties (called weak completeness and eventual weak accuracy). Consensus protocols based on such failure detectors are described in [4, 13, 17].

Usually, at some abstraction level, a system can be perceived as a set of (non-intersecting) groups, each group being composed of a set of processes (for example, a group can be a set of replicas implementing a fault-tolerant object). In such a system, the adequate communication primitive is the Multicast to Multiple Groups. While the target of a broadcast primitive is the set of all processes composing the system, the target of a multicast is limited to the processes belonging to a set of groups, distinct multicasts possibly having distinct targets. So, the target of a multicast is a set of groups dynamically defined by a parameter of the multicast (by a field of the multicast message).

As for broadcast, according to the properties related to the sets of delivered messages and to the order of message deliveries, several multicast primitives can be defined [8]. Here, we are interested in the Total Order Multicast primitive defined in the following way. Let $m < m'$ if a process delivers $m$ before $m'$. Then the message delivery relation “<” must be acyclic.

Total order multicast to multiple groups is particularly interesting either to implement data consistency criteria such as linearizability [12] or normality [10] (which allows an operation to perform on several objects), or to implement a specific class of transactions. Let us consider the following example taken from [18]. Consider a classical transaction that transfers $1,000$ from bank account #1 to bank account #2. To achieve fault-tolerance, assume that each bank account is replicated on several nodes, and assume that every replica is managed by a process. Let $g_1$ be the fault-tolerant group of processes that manage bank account #1, and let $g_2$ be the fault-tolerant group of processes that manage bank account #2. The two operations (withdrawal and deposit) can be aggregated into a single message by defining $m$ as: (remove $1,000$ from account #1; add $1,000$ to account #2). When a process in $g_1$ delivers $m$, it removes $1,000$ from the bank account it man-

---

*This author is supported by a grant of the CNPq/Brazil.
† Author names are listed in alphabetical order.
ages; when a process in $g_2$ delivers $m$, it adds $1,000 to the bank account it manages. In this distributed setting, the money transfer transaction can be expressed as the total order multicast of $m$ to the groups $g_1$ and $g_2$. It is easy to see that the total order property of multicast ensures the serializability of transactions.

A simple way to implement total order multicast to multiple groups is to use total order broadcast. Each message is sent with the total order broadcast primitive, and only processes belonging to a destination group are required to deliver it. This implementation is particularly inefficient. In a transaction system, such an approach would require that the implementation of any transaction accesses all the objects. So, the notion of genuine implementation of total order multicast has been introduced in [11]: an implementation is genuine if it satisfies the following Minimality property: “the only processes involved in the implementation of a total order multicast are the sender of the message and the processes belonging to destination groups”. The brute force implementation based on total order broadcast obviously does not satisfy this Minimality property. It has been shown in [11] that it is not possible to design a genuine implementation of total order multicast to multiple groups when both groups and failure detectors are unreliable. That is why, in the following, we consider that groups are reliable (i.e., in each group, there is a majority of processes that do not crash).

In this paper, we are interested in designing a protocol implementing total order multicast to multiple groups, in the context of asynchronous distributed systems where processes may crash. This implementation has two noteworthy properties. The first is the previous Minimality property. The protocol is based on the following principle. Each group has a logical clock to generate timestamps for the messages it receives. When they receive a message $m$, processes of a group $g_a$ use a consensus protocol as a sub-protocol to associate a single timestamp with $m$; this is the timestamp proposed by the group $g_i$ for $m$. Then, the groups that are destinations of $m$ exchange their proposals and compute the maximal one which becomes the definitive timestamp associated with $m$. Then, within each group, processes execute a second consensus protocol to consistently update their logical clocks that locally implement the clock of the group. Finally, a message can be delivered by a process as soon as it has the lowest timestamp among all undelivered messages. If each group reduces to a single reliable process, the proposed protocol reduces to the well-known Skeen’s protocol (described in [2]). On the other hand, if the system is composed of a single group, the proposed protocol reduces to a protocol close to the one designed by Chandra and Toueg for total order broadcast [4].

The second noteworthy property of the protocol is a Locality property. From a scalability point of view, a protocol in which each consensus is limited to a single group is preferable to a protocol in which several groups are involved in a consensus. So, we say that a consensus-based group protocol has the Locality property\(^1\) if no of its underlying consensus involves several groups. Such an approach favors a hierarchical decomposition of the problem and a modular implementation (each group can have its own consensus protocol). This is particularly attractive to implement total order multicast to multiple groups in large scale distributed systems. So, in asynchronous systems where communications are reliable and where processes can crash, the proposed protocol works when a consensus protocol works in each group taken individually.

The implementation of a multicast to multiple groups primitive has been addressed in several works. We review here a few of them. The Isis system [2] implements a weaker primitive, namely Local Total Order multicast (the relation “$<$” on message deliveries has not to be acyclic; only its projections on each group have to be acyclic). Total order multicast protocols implemented in TOTEM [1] and in TRANSIS [5] do not satisfy the Minimality property. The total order multicast protocol proposed in [7] assumes a perfect failure detector.

This paper is composed of seven sections. First Section 2 defines Total Order Multicast to Multiple Groups in asynchronous systems. Then, Section 3 presents the two underlying building blocks on top of which the proposed protocol is built (namely, Uniform Reliable Multicast and Uniform Consensus). Then, Section 4 presents the basic principles of the protocol and describes it. This protocol satisfies the Locality and the Minimality properties. Section 5 proves the protocol works. Section 6 describes an improvement that can considerably reduce the cost of the protocol.

2 Total Order Multicast

2.1 Asynchronous Systems and Groups

We consider a system composed of a finite set $\Pi$ of processes $p_1, p_2, \ldots, p_n$. A process can fail by crashing, i.e., by prematurely halting. A process behaves correctly (i.e., according to its specification which is defined by its program text) until it (possibly) crashes. By definition a correct process is a process that never crashes. Moreover, a crashed process remains crashed forever (Section 6.2 considers a crash/recovery model).

Processes communicate and synchronize by exchanging messages through communication channels. Every pair of processes is connected by a channel whose reliability is defined in the following way: a message sent by a process $p_i$ to a process $p_j$ is eventually received by $p_j$ if $p_i$ and $p_j$ are correct. Processes communicate by exchanging messages. The absence of a fixed bound on process relative speed and on message transfer delays makes the system asynchronous.

The set $\Pi$ of processes is statically partitioned into non-empty non-intersecting groups $g_1, g_2, \ldots, g_k$ (i.e., $\forall i : g_i \neq \emptyset$, $\forall i \neq j : g_i \cap g_j = \emptyset$ and $\bigcup_i g_i = \Pi$). How and why these groups are defined depends on the structure of the system or on the needs of upper layer applications. From the point of view of this paper, these points are irrelevant, we only consider groups do exist.

2.2 Total Order Multicast to Multiple Groups

We suppose all messages are distinct\(^2\). Let $m$ be a message. Its field named $m\text{dest}$ denotes the non-empty set of groups to which $m$ is sent. This field is filled in with group names by the sender before it sends $m$.

\(^1\)We borrow the term Locality from [12] where, albeit in a different context, it is used with a similar “modularity” meaning.

\(^2\)This can be easily ensured by adding an identity to each message, an identity being a pair (sequence number, sender identity).
Total Order Multicast to Multiple Groups (in short TO-multicast) is defined by two primitives, namely, \texttt{TO\_multicast}(m) and \texttt{TO\_deliver}(m). \texttt{TO\_multicast}(m) allows a process to send a message \( m \) to the processes of each group belonging to \( m_{dest} \). \texttt{TO\_deliver}(m) allows a process to deliver the message \( m \) sent by the invocation \texttt{TO\_multicast}(m). As in [8], when a process executes \texttt{TO\_multicast}(m) (resp. \texttt{TO\_deliver}(m)) we say that it “TO-multicasts” \( m \) (resp. “TO-delivers” \( m \)). The semantics of these primitive is defined by the following four properties\(^3\) [8]:

**Uniform Validity** If a process \( p \) TO-delivers \( m \), then some process has TO-multicast \( m \) and \( p \) belongs to a group \( g \) such that \( g \in m_{dest} \). This property defines the value domain of a delivered message (no spurious messages).

**Uniform Integrity** A process TO-delivers a message \( m \) at most once (no message duplication).

**Termination** If (1) a correct process TO-multicasts \( m \), or if (2) a process TO-delivers \( m \), then all correct processes that belong to another group of \( m_{dest} \) TO-deliver \( m \). This property defines the situations in which the multicast must terminate, i.e., the message \( m \) must eventually be delivered to its correct destination processes. There are two such situations. The first one is when the sender is correct (it has executed \texttt{TO\_multicast}(m) without crashing). The second one is when the message has been TO-delivered by a process. Said another way, the only case in which a multicast can not terminate is when the sender process crashes (e.g., during its invocation of \texttt{TO\_multicast}(m)).

**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. 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.

3 Underlying Protocols and Failures Related Assumptions

The protocol presented in Section 4 provides an implementation of TO-multicast that satisfies both the Minimality and the Locality properties. It is based on two building blocks, each of them solving a particular problem, namely, Uniform Reliable Multicast and Uniform Consensus. This section presents these two problems and states the additional assumptions that have to be satisfied in order these problems can be solved.

3.1 Uniform Reliable Multicast

Uniform Reliable Multicast to Multiple Groups is defined by two primitives [8]: \texttt{R\_multicast}(m) and \texttt{R\_deliver}(m). The semantics of these primitives is defined by the first three properties of TO-multicast, namely, Uniform Validity, Uniform Integrity

\(^3\)Some properties are qualified as being uniform [8]. A process mentioned in an uniform property can crash later. Conversely, a process mentioned in the same but non-uniform property is always supposed to be correct (never crashes). Here, we consider only uniform properties. From a practical point of view this is particularly meaningful: until it (possibly) crashes any process is a priori considered as being potentially correct.
crashed process or by suspecting a not crashed one. By restricting
the domain of mistakes they can make, several classes of failure
detectors can be defined. From a formal point of view, a failure
detector class is defined by two properties: a liveness property,
called Completeness, which addresses detection of actual failures,
and a safety property, called Accuracy, which restricts the mis-
takes a failure detector can make. Among the classes of failure
detectors defined by Chandra and Toueg, the class $\mathcal{S}$ is char-
terized by the two following properties. Its liveness property is
called Strong Completeness: Eventually, every crashed process is
permanently suspected by every correct process. Its safety prop-
erty is called Eventual Weak Accuracy: There is a time after which
some correct process is never suspected. It has been shown in [3]
that, provided a majority of processes are correct, these conditions
are the weakest ones to solve the consensus problem\footnote{0}. Consen-
sus protocols based on unreliable failure detectors of the class $\mathcal{S}$
have been proposed in [4, 13, 17]. So, in the following we suppose
that:

- Each group is equipped with a failure detector of the class $\mathcal{S}$.
- Let $f_i$ be the maximum number of processes of the group $g_i$
  that can crash. We assume that $\forall i : f_i < \lceil g_i \rceil/2$: a majority
  of processes are correct in each group. (This is why groups are
  qualified “reliable” in the Introduction.)

Note that each group can use a distinct consensus protocol. As
previously noted, this is particularly attractive for scalability when
implementing total order multicast in large scale distributed sys-

4 The Protocol

4.1 Underlying Principles

Our TO-multicast protocol borrows its basic principles (1)
from the Lamport’s clock protocol [15] (and from its adaptation
to the TO-multicast protocol designed by Sken for failure-free
systems\footnote{0}), and (2) from the Chandra-Toueg’s total order broadcast
protocol described in [4].

The protocol associates a timestamp with each message and
delivers messages according to the order defined by their times-
tamps. Each group is equipped with a clock with which it can
generate timestamps. A timestamp is a pair (clock value, group
identity). The timestamp associated with a message $m$ is denoted
$m.t$. The protocol proceeds in four consecutive steps. Let $m$ be a
message that has been TO-multicast to the set of groups defined by
$m.dest = \{g_2, g_3, \ldots\}$.

Step 1. Each group $g_x \in m.dest$ defines a timestamp for $m$ (de-
noted $m.t_x$). This timestamp is $g_x$’s proposal to be the definitive
timestamp for $m$.

Step 2. Each group of $m.dest$ proposes its timestamp for $m$ to
the other groups of $m.dest$. Then each group computes the greatest
timestamp proposed for $m$. Let $m.ts$ be this greatest times-
tamp: it is the definitive timestamp associated with $m$. (Note that,
by construction, it is the same for all groups.)

Step 3. Each group updates its clock according to the clock value of
$m.ts$.

Step 4. Finally, a group delivers $m$ when $m.ts$ is the lowest
timestamp among all the timestamped messages it has not yet de-
ivered (be these timestamps proposals or definitive timestamps).

To implement the previous principle, each process $p_i$ of a group
$g_x$ is endowed with (1) a local clock variable, $clock_i$, initialized to
0, that represents its view of the clock of the group $g_x$, and with
(2) a queue $Rec_i$ initially empty. When a message $m$ sent to $g_x$
is received by a process $p_i \in g_x$, it is stored at the tail of $Rec_i$. Mes-
 sage $m$ remains in this queue until it is TO-delivered. Its progress
towards TO-delivery is expressed by a linear automaton. More
precisely, each message has a field $m.state$ whose meaning is the
following:

$m.state = \emptyset$ means that $m$ has not yet been assigned a times-
tamp (initially, for any message $m$, $m.state = \emptyset$).
$m.state = \{q_0\}$ means that $m$ has been assigned a timestamp by
the group $g_x (m.ts_x)$.
$m.state = \{q_1\}$ means that $m$ has its final timestamp ($m.ts$).
$m.state = \{q_2\}$ means that the clock of $g_x$ (for $p_i$, this clock is lo-
 cally represented by $clock_i$) has been resynchronized with respect
to the clock value of the definitive timestamp of $m$ (namely, $m.ts$).

4.2 Description of the Protocol

Implementation of $TOMulticast(m)$. The implementation of $TOMulticast(m)$ is easy. When a process $p$ invokes $TOMulticast(m)$ it issues a call to the uniform reliable
multicast primitive, i.e., it calls $Multicast(m)$. As indicated in
Section 3.1 this ensures that if $p_i$ is correct or if a process (be-
onging to a group $g_x \in m.dest)$ R-delivers $m$, then all correct
processes that belong to groups of $m.dest$ eventually R-deliver $m$.

Remark. Note that in the implementation of $Multicast(m)$, when a
process receives a message for the first time, it needs to forward this
message (1) to all processes of its group and (2) only to a majority of processes
of each other group of $m.dest$. As by assumption, in each group, there
is at least one correct process in any majority, the Termination property of
uniform reliable multicast is ensured. End of remark

This shows that the implementation of the TO-multicast
primitive is simple. However, as we will see in the next parts of this
section, the implementation of the TO-deliver primitive is far from
being trivial: we have to build the Global Total Order property that
distinguishes uniform reliable multicast from TO-multicast.

Structure of a process. In addition to the execution of the application program it is as-
sociated with, each process $p_i$ of a group $g_x$ is made up of sev-
eral threads. A thread $T^p_i$ is associated with each message $m$ R-
delivered but not yet TO-delivered to $p_i$. Moreover, a permanent
thread $T^p_{clus}$ ensures that $clock_i$ (the local clock of $p_i$) provides
$p_i$ with a correct implementation of the clock of the group $g_x$. Its
role is fundamental. It guarantees that each process $p_i$ of a group $g_x$ provides the same timestamps to the same messages, and consequently ensures that all processes of a group $g_x$ offer to the other groups the same view of $g_x$’s clock. The next two paragraphs describe these two types of threads that (in addition to the local user program) compose a process $p_i$.

To ease explanations, we assume that, at any time, among the set of threads $T^m_{i}$ and the thread $T^{cons}_{i}$ of $p_i$, at most one of them is active. Moreover, except when it executes a wait until statement, a thread executes atomically (it cannot be interrupted). This ensures local consistency of the local variable $Rec_i(clock)$ is accessed only by $T^{cons}_{i}$.

figure 1. local thread $T^m_{i}$ associated with the processing of a message $m$

Thread associated with the processing of a message.

When a message $m$ is R-delivered at a process $p_i$ belonging to a group $g_x$, this process creates and associates with $m$ a new thread whose behavior is defined in Figure 1. This thread $T^m_{i}$ first initializes $m\_state$ to $q_0$ and adds $m$ to the tail of $Rec_i$ (line 1.1). Then, as indicated previously, $T^m_{i}$ progresses toward the TO-delivery of $m$ by executing the four steps mentioned in Section 4.1. This progress of $m$ is “measured” by its field $m\_state$ (which takes the successive values $q_0, q_1, q_2$, and finally $q_3$).

Step 1. (Line 1.2). $T^m_{i}$ first blocks until $m$ has been assigned a group timestamp $m\_ts^g_i$ (the group to which the concerned process $p_i$ belongs: $p_i \in g_x \land g_x \in m\_dest$) (line 1.2). This timestamp is computed by the set of threads $T^{cons}_{i}$ of processes $g_x$, as described in the next paragraph (Figure 2). $T^m_{i}$ knows $m$ has got this timestamp when it discovers that $m\_state$ has taken the value $q_2$. (Recall that a timestamp is a pair (clock value, group identity).)

Step 2. (Lines 1.3-1.5). Then, $T^m_{i}$ sends this timestamp $m\_ts^g_i$ to all processes belonging to groups appearing in $m\_dest$ (line 1.3). $m\_ts^g_i$ is $g_x$’s proposal to compute the definitive timestamp that will be assigned to $m$. $T^m_{i}$ then waits until it has received a timestamp proposal $m\_ts^g_j$ from each group $g_y$ of $m\_dest$ (line 1.4): this is done as soon as it has received a timestamp from at least one process of each group $g_y$ of $m\_dest$. After having received all these values, $T^m_{i}$ computes the definitive timestamp for $m$ (denoted $m\_ts$): it is the greatest timestamp proposed for $m$ by its destination groups. As soon as $m$ has got its final timestamp, $T^m_{i}$ makes its state evolve to $q_3$ (line 1.5).

Step 3. (Line 1.6). Then, $T^m_{i}$ waits until the clock of the group $g_x$ (locally represented by $clock_x$) has been resynchronized to $max(clock_x, clock_{value of m\_ts}$). This resynchronization is not managed by $T^m_{i}$: it is done by the thread $T^{cons}_{i}$ described in Figure 2. $T^m_{i}$ learns this clock synchronization due to $m$ is done by discovering that $m\_state = q_3$ (line 1.6). Note that, as soon as $m\_state$ is set to $q_3$, all messages $m'$ in $Rec_i$ such that $m'\_state = q_3$ will get a timestamp greater than $m\_ts$.

Step 4. (Lines 1.7-1.8). When $m$ has the lowest timestamp with respect to all the messages $m'$ in $Rec_i$ such that $m'\_state = q_1$ or $q_2$, then $T^m_{i}$ TO-delivers $m$ (line 1.8). Moreover the thread $T^m_{i}$ is killed.

Remark. If a message $m$ is TO-Multicast to a single group $g_x$, then the thread $T^m_{i}$ can be simplified. The lines 1.3-1.6 can be replaced by the single statement $m\_state \leftarrow q_3$; as $m$ is addressed only to $g_x$, it can skip steps 2 and 3 as these steps are due to the multiplicity of destination groups. Moreover, in the extreme case where the system is composed of a single group, additional simplifications can be done: the message state $q_2$ can be suppressed and the message states $q_1$ and $q_3$ can be merged. These simplifications result in a TO-broadcast protocol whose behavior is close to the consensus-based TO-broadcast protocol proposed by Chandra and Toueg in [4]. End of remark

Thread associated with the management of a group clock.

The set of threads $T^{cons}_{i}$ of processes $p_i \in g_x$ play a crucial role with respect to the group $g_x$. They implement the clock of the group $g_x$. This clock, represented by $clock_{i}$, is used by $g_x$ to define a timestamp ($m\_ts^g$) for each message R-delivered to the group $g_x$. (Note that $T^m_{i}$ does not access $clock_{i}$.)

This thread is described in Figure 2. Its core is a consensus protocol executed within $g_x$. Let us consider the situation where each non-crashed process $p_i \in g_x$ has been R-delivered one or several messages. So, at each $p_i$ we have ($3m \in Rec_i: m\_state = q_0$). In this situation, the thread $T^{cons}_{i}$ of any non-crashed process $p_i \in g_x$, increases its local variable $clock_{i}$ (line 2.3) let $m$ be the new value of $clock_{i}$ then launches an uniform consensus within $g_x$ by proposing a message (line 2.4). All executions of consensus within $g_x$ are identified by a clock value. So, as in [4], within $g_x$ the consensus execution numbered $k$ is tagged with the clock value $m$ and the corresponding primitives are propose($k, -$) and decide($k, -$) (line 2.4). The aim of the consensus execution numbered $k$ is either to associate a timestamp $(k, x)$ (where $x$ is the identity of $g_x$) with some message M R-delivered to $g_x$ ("timesamping consensus") or (2) to entail a resynchronization of the clock of $g_x$ ("resynchronization consensus").

Figure 2. Local Thread $T^{cons}_{i}$ implementing the Clock of Group $g_x$
Let us consider both cases. $T_i^{con}$ has proposed $m_i$ to the consensus execution numbered $k$, and the result of this consensus is some message $m'$ (line 2.4). According to the state of the message $m'$, that is output by the consensus numbered $k$, this consensus is either a "timestamping consensus" (when $m'.state = q_0$, line 2.6) or a "resynchronization consensus" (when $m'.state = q_2$, line 2.7).

- We first consider case (1): $m'.state = q_0$. In this case, $m'$ has been proposed by some process of $g_2$ to get a group timestamp $m'.ts^k$. So, $m'$ gets one, namely $(k, x)$, and $m'.state$ is updated to $q_1$ (line 2.6). Note that from the Uniform Agreement property of consensus, the pair $(k, x)$ is consistently considered as the single value of $m'.ts^k$ by every non-crashed process $p_i \in g_2$. The group acts as if it had used a single clock to timestamp $m'$. This is the case where the consensus gives a group timestamp to a message. That is why we call it "timestamping consensus".

- Let us consider case (2): $m'.state = q_2$. As we have seen is Section 4.1, a group $g_j$ has to resynchronize its clock (perceived by $p_i$ as clock($i$)) after it has learnt that a message $m$ has got its definitive timestamp $m.ts$. From the point of view of $p_i$, this resynchronization is expressed as $clock_i \leftarrow max(clock_i, clock$ of $m.ts)$. To ensure all $(clock_i)$ variables consistently implement the clock of $g_j$, they must take the same sequence of values: if a local clock progresses from $k$ to $k + 1$, the other local clocks of $g_j$ have to progress from $k$ to $k + 1$; if a clock jumps from $k$ to $k'$, the other clock variables must also jump from $k$ to $k'$. In other words, if local clocks of processes of $g_j$ are equal before being resynchronized (let $k$ be their value), the resynchronization will entail the same jump for all of them, namely, they will jump from $k$ to max($k$, clock value of $m.ts$). And, consequently, all local clocks of a group will take the same sequence of values.

A way to ensure all local clocks have the same value before being resynchronized, is to do the resynchronization immediately after a consensus: just after the consensus numbered $k$, all clocks are equal to $k$. That is why, within a group, processes propose to consensus messages that require this group to resynchronize its clocks, i.e., they propose messages $m'$ such that $m'.state = q_2$ (lines 2.1 and 2.4). When such a message $m'$ with $m'.state = q_2$ is output by consensus numbered $k$, i.e., $T_i^{con}$ executes decide($k, m'$) at line 2.4), $T_i^{con}$ does the corresponding clock resynchronization (line 2.7). That is why when the result of a consensus execution is a message $m'$ such that $m'.state = q_2$, the consensus is called "resynchronization consensus".

To sum up, let us consider a message $m$ proposed by $T_i^{con}$ to consensus numbered $k$.

- If $m.state = q_0$, then $T_i^{con}$ proposes $m$ for it to obtain a group timestamp. Message $m$ will get this timestamp when it will be output by some consensus numbered $k'$. The clock value of $m.ts^k$ will be the number $k'$ ("timestamping consensus").

- If $m.state = q_2$, then $T_i^{con}$ proposes $m$ for all the clocks of the group $g_i$ resynchronize consistently with respect to the clock value of the definitive timestamp $m.ts$. This resynchronization occurs when $m$ is output by a consensus ("resynchronization consensus").

5 Correctness Proof

A simple examination of the protocol shows that it satisfies the Locality property and the Minimality property. Due to space limitation the proof that the protocol satisfies the properties stated in Section 2.2 is omitted. The reader interested in the proof may consult [6].

6 Discussion

6.1 Reducing the Consensus Cost

When considering the proposed protocol, the cost of TO-multicasting a message is the addition of the cost of uniform reliable multicast, plus the cost of consensus, plus the cost of step 2 (the exchange of group timestamps to compute the greatest one). Here, we shows how this cost can be reduced.

In Section 4, the proposal of a thread $T_i^{con}$ to a consensus execution is composed of a single message. Actually, it is possible to modify this protocol in order a thread $T_i^{con}$ be allowed to propose several messages to a given consensus. Now, the size of a proposal is not reduced to a single message, it may include a set of messages $MSG = \{m_1, \ldots, m_j\}$ such that some of them are in state $q_0$ and others are in state $q_2$. So, within a group $g_x$, the result of the consensus numbered $k$ can be a set of messages $MSG' = \{m'_1, m'_2, \ldots, m'_k\}$. Each message $m$ is required to carry its identity $m.id$, and message identities must be such that two distinct messages must have distinct but comparable identities.

When a thread $T_i^{con}$ executes decide($k, MSG'$) with $MSG' = \{m'_1, m'_2, \ldots, m'_k\}$, it has to sequentially do the following actions:

- Let $MSG_0 = \{m' \in MSG' : m'.state = q_0\}$. If $MSG_0 \neq \emptyset$, then $\forall m' \in MSG_0, T_i^{con}$ associates with $m'$ the following group timestamp: $(k, x, m'.id)$. So, a timestamp is now made of three fields: a clock value, a group identity and the message identity. As several messages can now have the same first two timestamp fields, it is necessary to add the third field in order different messages have different timestamps. Note that these timestamps are lexicographically ordered, and consequently they define a total order.

- Let $MSG_2 = \{m' \in MSG' : m'.state = q_2\}$. If $MSG_2 \neq \emptyset$, $T_i^{con}$ computes the highest clock value $k'$ among the set of timestamps of messages $\in MSG_2$. Then, each $T_i^{con}$ resynchronizes its clock, local variable to max(clock, $k'$), i.e., clock is directly resynchronized to the highest clock value without going through the other resynchronization values. (This is due to the fact that clock resynchronizations are only required to ensure the liveness property -termination- of the protocol.)

When applied within each group, these modifications can considerably reduce the number of consensus executions. Hence, the cost of the original protocol can be reduced at the price of a small increase in the timestamp size.

---

6 As it is done in the total order broadcast protocol proposed in [4].

7 Message identities of the form (sequence number, sender identity) have the required properties.
6.2 Crash/Recovery Model

In a crash/recovery model, a process can crash and later recover. When it crashes, a process loses its context. To cope with this context loss, a process has to log critical data into stable storage. In this model, a correct process is a process that eventually does not crash anymore. Moreover, messages arriving at a destination process while it was crashed are lost. A consensus protocol suited to such a fault model is described in [14].

The protocol proposed in Section 4 works in a crash/recovery model provided that: (1) each group be provided with a consensus process while it was crashed are lost. A consensus protocol can be reduced. Moreover, messages arrived at a destination have been shown how the consensus cost of the protocol still works if some consensus can be reduced. Moreover, messages arrived at a destination property of the multicast protocol.

7 Conclusion

This paper addressed the Total Order Multicast to Multiple Groups problem in the context of asynchronous distributed systems in which processes may suffer crash failures. “Multicast to Multiple Groups” means that a message is sent to a subset of the process groups composing the system, distinct messages possibly having distinct destination groups. “Total Order” means that all message deliveries must be totally ordered.

A protocol for such a multicast primitive has been proposed. This protocol uses two underlying building blocks, namely, Uniform Reliable Multicast and Uniform Consensus. As we have seen, this protocol enjoys two noteworthy properties: Minimality and Locality. Minimality means that only the sender of a message and processes of its destination groups do participate in its multicast. Locality means that any consensus execution is restricted to processes of a single group. These properties are particularly useful when one is interested in using a total order multicast primitive in large scale distributed systems. Lastly, it has been shown how the consensus cost of the protocol can be reduced.

References


