Consensus-Based Fault-Tolerant Total Order Multicast

Udo Fritzke Jr., Philippe Ingels, Achour Mostefaoui, and Michel Raynal

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 investigates a consensus-based approach to solve this problem and proposes a corresponding protocol to implement this 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 total order 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 is 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. In addition to a correctness proof, an improvement that reduces the cost of the protocol is also suggested.

Index Terms—Asynchronous systems, consensus, groups, reliable multicast, total order, group multicast.

1 INTRODUCTION

Total Order Broadcast (also called Atomic Broadcast) is one of the most important agreement problems encountered in the design and implementation of fault-tolerant distributed systems. This problem consists of 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 [19]). 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., noncrashed) copies of a service are delivered at the same set of updates in the same order and, consequently, the state of the service is kept consistent [23].

It has been shown in [5] 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 [8] 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 [5], [16], [18], [21].

Usually, at some abstraction level, a system can be perceived as a set of (nonintersecting) 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 (or, equivalently, 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 [14]. 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 (this primitive is called Global Atomic Multicast in [14]).

Total order multicast to multiple groups is particularly interesting either to implement data consistency criteria, such as linearizability [15] or normality [11] (which allows an operation to be on several objects) or to implement a specific class of transactions. Let us consider the following example taken from [22]. Consider a classical transaction that transfers $1,000 from bank account #1 to bank account #2. To achieve fault tolerance, assume that each

---

*The authors are with IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France. E-mail: {ufritzke, ingels, achour, raynal}@irisa.fr.
For information on obtaining reprints of this article, please send e-mail to: tpsd@computer.org, and reference IEEECS Log Number 112983.
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: \( (\text{remove } $1,000 \text{ from account } #1; \text{add } $1,000 \text{ to account } #2) \). When a process in \( g_1 \) delivers \( m \), it removes $1,000 from the bank account it manages; 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 property 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; for instance, 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 [12]. 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 [12] 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 are a majority of processes that do not crash).

In this paper, we consider asynchronous distributed systems where processes may crash and investigate the use of consensus to implement a Total Order Multicast to Multiple Groups primitive in such a context. This proposed 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 \( g_y \) use a consensus protocol as a subprotocol to associate a single timestamp with \( m \); this is the timestamp proposed by the group \( g_y \) 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 [5].

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 when none of its underlying consensus operations involve multiple groups (the term Locality is borrowed from [15] where, albeit in a different context, it is used with a similar “modularity” meaning). 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 [6]. Here, we review a few of them. Like the protocol proposed in [20], the ISIS system [3] implements a weaker primitive, namely, Local Total Order multicast (the relation “<” on message deliveries need not be acyclic; only its projections on each group have to be acyclic). Total order multicast protocols implemented in TOTEM [1] and in TRANSIS [7] do not satisfy the Minimality property. The total order multicast protocol proposed in [10] assumes a reliable failure detector. Finally, the protocol presented in [13] uses an underlying communication layer providing causal message delivery; so, it does not satisfy the Minimality property; moreover, it also fails to meet the Locality property and does not ensure the acyclicity of “<” in all scenarios.

This paper is comprised of eight 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. These building blocks are 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 (provided a consensus protocol works in each group). Section 6 describes an improvement that reduces the cost of the protocol. Finally, Section 7 describes two particular instances of the protocol and Section 8 concludes the paper.

2 TOTAL ORDER MULTICAST IN ASYNCHRONOUS SYSTEMS

2.1 Asynchronous Systems and Groups

We consider a system comprised of a finite set 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.
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. The multiplicity of processes and message-passing communication make the system distributed. There is no bound on process relative speed and on message transfer delays. This absence of timing assumptions makes the system asynchronous.

The set \( I \) of processes is statically partitioned into nonempty nonintersecting 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 \( \cup g_i = I \)). 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 which do exist.

### 2.2 Total Order Multicast to Multiple Groups

We assume that all messages are different (this can be easily ensured by adding an identity to each message, an identity being a pair (sequence number, sender identity)). Let \( m \) be a message. Its field, named \( m_{\text{dest}} \), denotes the nonempty set of groups to which \( m \) is sent. This field is filled in with group names by the sender before it sends \( m \).

**Total Order Multicast to Multiple Groups (in short TO-multicast)** is defined by two primitives, namely, \( \text{TO\_multicast}(m) \) and \( \text{TO\_deliver}(m) \). \( \text{TO\_multicast}(m) \) allows a process to send a message \( m \) to the processes of each group belonging to \( m_{\text{dest}} \). \( \text{TO\_deliver}(m) \) allows a process to deliver the message \( m \) sent by the invocation \( \text{TO\_multicast}(m) \). As in [14], when a process executes \( \text{TO\_multicast}(m) \) (resp. \( \text{TO\_deliver}(m) \)), we say that it “TO-multicasts” \( m \) (resp. “TO-delivers” \( m \)). The semantics of these primitives are defined by the following four properties\(^1\) [14]:

- **Uniform Validity.** If a process \( p \) TO-delivers \( m \), then some process has TO-multicasted \( m \) and \( p \) belongs to a group \( g \) such that \( g \in m_{\text{dest}} \).

  This property expresses that there are no spurious messages; it defines the value domain of a delivered message.

- **Uniform Integrity.** A process TO-delivers a message \( m \) at most once.

  This property expresses that there is no 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 a group of \( m_{\text{dest}} \) 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 (case 1) is when the sender is correct (in that case, it executed \( \text{TO\_multicast}(m) \) without crashing). The second one (case 2) is when the message has been TO-delivered by a process. Said another way, the only case in which a multicast cannot terminate is when the sender process crashes (e.g., during its invocation of \( \text{TO\_multicast}(m) \)).

- **Global Total Order.** Let \( \prec \) be the relation on messages defined in the following way: If a process TO-delivers \( m_1 \) before \( m_2 \), then \( m_1 \prec m_2 \). The relation \( \prec \) is acyclic.

  This property expresses that the set of delivered messages can be totally ordered (by doing a topological sort of \( \prec \)) in a way consistent with the message delivery order at each process. From a “logical” point of view (e.g., the point of view of an external observer), this means that it is possible to consider that the messages have been delivered one after the other, all deliveries of the same message occurring “simultaneously.”

### 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 for these problems can be solved.

#### 3.1 Uniform Reliable Multicast

Uniform Reliable Multicast to Multiple Groups is defined by two primitives [14]: \( R_{\text{multicast}}(m) \) and \( R_{\text{deliver}}(m) \). The semantics of these primitives are defined by the first three properties of TO-multicast, namely, uniform validity, uniform integrity, and termination, where the primitive identifiers \( \text{TO\_multicast} \) and \( \text{TO\_deliver} \) are replaced by \( R_{\text{multicast}} \) and \( R_{\text{deliver}} \), respectively. Said in another way, \( \text{TO\_multicast} = \text{Uniform Reliable Multicast} + \text{Global Total Order} \).

Implementation of Uniform Reliable Multicast is relatively easy. When a process receives a message \( m \) for the first time, it first forwards \( m \) to the processes belonging to groups in \( m_{\text{dest}} \) and only then considers the delivery of \( m \) [14]. As we will see in Section 4, adding a total order delivery constraint to uniform reliable multicast is not a trivial task. Moreover, when the system is comprised of a single group, multicast is reduced to broadcast and, as indicated in Section 1, Chandra and Toueg have shown in [5] that total order broadcast and consensus are equivalent problems.

#### 3.2 Uniform Consensus

As in our TO-multicast protocol, each instance of the consensus problem is local to a group (i.e., involves processes of a single group) and we use a single group to formulate this problem.

In the **Consensus** problem, each process proposes a value and all correct processes have to decide on some value \( v \) that is related to the set of proposed values [8]. Formally, the **Uniform Consensus** problem is defined in terms of two
primitives: propose and decide. As in previous works (e.g., [5]), when a process \( p \) invokes \( \text{propose}(w) \), where \( w \) is its proposal to the consensus, we say that \( p \) “proposes” \( w \). In the same way, when \( p \) invokes \( \text{decide} \) and gets \( v \) as a result, we say that \( p \) “decides” \( v \) (denoted \( \text{decide}(v) \)).

The semantics of propose and decide are defined by the following properties:

- **Uniform Validity.** If a process decides \( v \), then \( v \) was proposed by some process.
  
  This property defines the value domain of the result.
- **Uniform Integrity.** A process decides at most once.
  
  This property states there are no “duplicates.”

From the point of view of each process, there is a single decision.\(^2\)
- **Termination.** All correct processes eventually decide.
  
  This property states that at least all correct processes decide.
- **Uniform Agreement.** No two processes (correct or not) decide differently.
  
  This property gives its global meaning to the consensus: From the point of view of all processes, there is a single decision.\(^3\)

### 3.3 About Failures

As noted in Section 1, it has been shown by Fischer et al. [8] that the consensus problem has no deterministic solution in asynchronous distributed systems that are subject to even a single process crash failure. Intuitively, this negative result is due to the impossibility of safely distinguishing (in an asynchronous setting) a crashed process from a slow process (or from a process with which communications are very slow).

This impossibility result has motivated researchers to find a set of minimal assumptions that, when satisfied by a distributed system, makes consensus solvable in this system. Chandra-Toueg’s Unreliable Failure Detector concept constitutes an answer to this challenge [5]. From a practical point of view, an unreliable failure detector can be seen as a set of oracles: Each oracle is attached to a process and provides it with a list of processes it suspects to have crashed. An oracle can make mistakes by not suspecting a 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 mistakes a failure detector can make. Among the classes of failure detectors defined by Chandra and Toueg, the class \( \circ S \) is characterized by the two following properties: Its liveness property, called Strong Completeness, states that: Eventually, every crashed process is permanently suspected by every correct process.

Its safety property, called Eventual Weak Accuracy, states that: There is a time after which some correct process is never suspected. It has been shown in [4] that, provided a majority of processes are correct, these conditions are the weakest ones to solve the consensus problem. Consensus protocols, based on unreliable failure detectors of the class \( \circ S \), have been proposed in [5], [16], [18], [21]. So, in the following, we suppose that:

- Each group is equipped with a failure detector of the class \( \circ S \).
- Let \( f \) be the maximum number of processes of the group \( g \) that can crash. We assume that \( \forall i: f_i < |g_i|/2 \): A majority of processes are correct in each group. (This is why groups are qualified “reliable” in Section 1.)

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 systems.

## 4 The Protocol

### 4.1 Underlying Principles

Our TO-multicast protocol borrows its basic principles 1) from the Lamport’s clock protocol [17] (and from its adaptation to the TO-multicast protocol designed by Skeen for failure-free systems) and 2) from Chandra-Toueg’s total order broadcast protocol described in [5].

The protocol associates a timestamp with each message and delivers messages according to the order defined by their timestamps. 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 \( m.\text{ts} \).

The protocol proceeds in four consecutive steps. Let \( m \) be a message that has been TO-multicast to the set of groups defined by \( m.\text{dest} = \{g_1, g_2, \ldots \} \).

**Step 1.** Each group \( g \in m.\text{dest} \) defines a timestamp for \( m \) (denoted \( m.\text{ts}^g \)). This timestamp is \( g \)'s proposal to be the definitive timestamp for \( m \).

**Step 2.** Each group of \( m.\text{dest} \) proposes its timestamp for \( m \) to the other groups of \( m.\text{dest} \). Then, each group computes the greatest timestamp proposed for \( m \). Let \( m.\text{ts} \) be this greatest timestamp; 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 \( m.\text{ts} \).

**Step 4.** Finally, a group delivers \( m \) when \( m.\text{ts} \) is the lowest timestamp among all the timestamped messages it has not yet delivered (be these timestamps proposals or definitive timestamps).

To implement the previous principle, each process \( p \) of a group \( g \) is endowed with 1) a local clock variable, \( \text{clock}_i \), initialized to 0 that represents its view of the clock of the group \( g \), and with 2) a queue \( \text{Rec}_i \), initially empty. When a message \( m \) sent to \( g \) is received by a process \( p \in g \), it is stored at the tail of \( \text{Rec}_i \). Message \( m \) remains in this queue until it is TO-delivered. Its progress toward TO-delivery is expressed by a linear automaton. More precisely, each
message has a field $m\text{.state}$ whose meaning is the following one:

- $[m\text{.state} = q_0]$ means that $m$ has not yet been assigned a timestamp (initially, for any message $m$, $m\text{.state} = q_0$).
- $[m\text{.state} = q_1]$ means that $m$ has been assigned a timestamp by the group $g_x$ ($m.txs$).
- $[m\text{.state} = q_2]$ means that $m$ has its final timestamp ($m.txs$).
- $[m\text{.state} = q_3]$ means that the clock of $g_x$ (for $p_i$, this clock is locally represented by $\text{clock}_i$) has been resynchronized with respect to the clock value of the definitive timestamp of $m$ (namely, $m.txs$).

4.2 Description of the Protocol

4.2.1 Implementation of TO_multicast($m$)

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

Remark. Note that, in the implementation of $R\text{.multicast}(m)$, when a process receives a message for the first time, it needs to forward this message 1) to all processes of its group of $m\text{.dest}$ and, finally, $q_3$.

Step 1 (Line 1.3). $T_i^m$ first blocks until $m$ has been assigned a group timestamp $m.txs$ ($g_x$ is the group to which the concerned process $p_i$ belongs: $p_i \in g_x \land g_x \in m\text{.dest}$ (line 1.3)).

This timestamp is computed by the set of threads $T_i^{cons}$ of processes $\in g_x$, as described in Fig. 2. $T_i^m$ knows $m$ has received this timestamp when it discovers that $m\text{.state}$ has taken the value $q_1$. (Recall that a timestamp is a pair (clock value, group identity).)

Step 2 (Lines 1.3-1.5). Then, $T_i^m$ sends this timestamp $m.txs$ to all processes belonging to groups appearing in $m\text{.dest}$ (line 1.3). $m.txs$ is $g_x$’s proposal to compute the definitive timestamp that will be assigned to $m$.

$T_i^m$ then waits until it has received a timestamp proposal $m.txs' \in g_x\text{.dest}$ from each group $g_y \in m\text{.dest}$ (line 1.4); this is done as soon as it has received a timestamp from at least one process of each group $g_y \in m\text{.dest}$. After having received all these values, $T_i^m$ 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 received its final timestamp, \(T_m^w\) makes its state evolve to \(q_3\) (line 1.5).

**Step 3** (Line 1.6). Then, \(T_m^w\) waits until the clock of the group \(g_x\) (locally represented by \(clock\)) has been resynchronized to \(\max(clock_i, \text{clock value of } m.ts)\). This resynchronization is not managed by \(T_m^w\); it is done by the thread \(T_{cons}^m\) described in Fig. 2. \(T_m^w\) learns this clock synchronization due to \(m\) is done by discovering that \(m.state = q_0\) (line 1.6).

Note that, as soon as \(m.state\) is set to \(q_3\), all messages \(m'\) in \(Rec\), such that \(m'.state = q_0\) 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 \(Rec\), such that \(m'.state = q_1, q_2\) or \(q_3\), then \(T_m^w\) TO-delivers \(m\) (line 1.8). Moreover, the thread \(T_m^w\) is killed.

**Remark.** If a message \(m\) is TO-Multicast to a single group \(g_x\), then the thread \(T_m^w\) can be simplified. 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 [5].

**4.2.4 Thread Associated with the Management of a Group Clock**

The set of threads \(T_{cons}^m\) 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 \(clock\), at \(p_i\), increases like a Lamport's clock [17] and is used by \(g_x\) to define a timestamp \((m.ts)\) for each message R-delivered to the group \(g_x\). (Note that \(T_m^w\) does not access \(clock\).)

This thread is described in Fig. 2. Its core is a consensus protocol executed within \(g_x\). Let us consider the situation where each noncrashed process, \(p_i \in g_x\), has R-delivered one or several messages. So, at each \(p_i\), we have \((\exists m \in Rec_i : m.state = q_0)\). In this situation, the thread \(T_{cons}^m\) of any noncrashed process \(p_i \in g_x\), increases its local variable \(clock_i\) (line 2.3). Let \(k\) be the new value of \(clock\), and launch a 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 [5], within \(g_x\), the consensus execution numbered \(k\) is tagged with the clock value \(k\) and the corresponding primitives are \(propose(k, -)\) and \(decide (k, -)\) (line 2.4). The aim of the consensus execution numbered \(k\) is either 1) to associate a timestamp \((k, x)\) (where \(x\) is the identity of \(g_x\)) with some message \(m\) R-delivered to \(g_x\) (“timestamping consensus”) or 2) to entail a resynchronization of the clock of \(g_x\) (“resynchronization consensus”).

Let us consider both cases. \(T_{cons}^m\) has proposed \(m\) 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 recommended by some process of \(g_x\) for this message to get a group timestamp \(m'.ts\). 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 \(m'.ts\) by every noncrashed process \(p_i \in g_x\). So, 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 in Section 4.1, a group \(g_x\) has to resynchronize its clock (perceived by \(p_i\) as \(clock_i\)) after it has learned that a message \(m\) has received its definitive timestamp \(m.ts\). From the point of view of \(p_i\), this resynchronization is expressed as \(clock_i \leftarrow \max(clock_i, \text{clock value of } m.ts)\). To ensure all \(clock_i\) variables consistently implement the clock of \(g_x\), they must take the same sequence of values: If a local clock progresses from \(k\) to \(k + 1\), the other local clocks of \(g_x\) 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 \( g_x \) 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 \( \max(k, \text{clock value of } m.ts) \). And, consequently, all local clocks of a group will take the same sequence of values.

A way to ensure that 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 recommend consensus messages that require this group to resynchronize its clocks, i.e., they recommend messages \( m' \) such that \( m'.state = q_0 \) (lines 2.1 and 2.4). When such a message \( m' \) with \( m'.state = q_0 \) is output by consensus numbered \( k \) (i.e., \( T_{cons}^o \) executes \( \text{decide}(k, m') \) at line 2.4), \( T_{cons}^o \) does the corresponding clock resynchronization (line 2.7).

That is why, when the result of a consensus execution is a message \( m' \) such that \( m'.state = q_0 \), the consensus is called "resynchronization consensus."

To sum up, let us consider a message \( m \) recommended by \( T_{cons}^o \) to consensus numbered \( k \):

- If \( m.state = q_0 \), then \( T_{cons}^o \) 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 \( m.ts^o \) will be the number \( k' \) ("timestamping consensus").
- If \( m.state = q_2 \), then \( T_{cons}^o \) proposes \( m \) for all the clocks of the group 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 stated in Section 1. It also satisfies the Minimality property, i.e., it provides a genuine [12] implementation of TO-multicast provided that the protocols implementing the underlying uniform reliable multicast and uniform consensus is genuine. The simple implementation of the \( R_{multicast}(m) \) and \( R_{deliver}(m) \) primitives sketched in Section 3.1 (and based on the uniform reliable broadcast protocol described in [14]) is genuine. Moreover, any uniform consensus protocol (such as the ones described in [5], [16], [18], [21]) can be used within a group, thus providing a genuine implementation.

5.1 Uniform Validity and Integrity

As threads \( T_i^m \) and \( T_{cons}^o \) neither create messages nor duplicate messages that have been R-delivered, Uniform Validity and Uniform Integrity of TO-multicast follow directly from the corresponding properties of the underlying uniform reliable multicast protocol that is used.

5.2 Termination

This section first establishes a set of lemmas which make modular the proof of the Termination property.

Lemma 1. 1) If a correct process TO-multicasts a message \( m \) or 2) if a process TO-delivers \( m \), then any correct process \( p_i \) belonging to a group \( g_x \in m.dest \) R-delivers \( m \) and launches a thread \( T_i^m \).

Proof. TO-multicast\((m)\) is implemented by a call to \( R_{multicast}(m) \). Moreover, let us note that if a process TO-delivers \( m \), it has previously R-delivered \( m \) (line 2.5). The lemma follows directly from the Termination property of the \( R_{multicast} \) issued by the sender of \( m \). More explicitly, if the sender is correct or if a destination process R-delivers \( m \), then any correct destination process \( p_i \) R-delivers \( m \) and, consequently, launches a thread \( T_i^m \). \( \Box \)

Lemma 2. Let us consider any two processes \( p_i \) and \( p_j \in g_x \). As long as none of them has crashed, if \( p_i \) executes \( \text{propose}(k, -) \), then \( p_j \) executes \( \text{propose}(k, -) \).

Proof. The proof is by induction on the number \( k \) (clock value) identifying each consensus executed within a group \( g_x \). Initially, all local clocks of processes of \( g_x \) are initialized to 0. So, the first consensus in which a noncrashed process of \( g_x \) participate is numbered 1 (lines 2.3-2.4).

As long as the output of the current consensus (numbered \( k \)) is a message \( m' \) such that \( m'.state = q_0 \), there is no clock resynchronization and the next consensus executed by processes of \( g_x \) is numbered \( k + 1 \). When the result of consensus numbered \( k \) is a message \( m' \) such that \( m'.state = q_2 \), then a clock resynchronization occurs (line 2.7). But, then, all noncrashed processes do the same update: Each is terminating consensus number \( k \) and immediately updates its \( clock_i \), local variable to the same value as the others, namely, to the value \( \max(k, \text{clock value of } m'.ts) \). It follows that all noncrashed processes of \( g_x \) execute the same sequence of numbered consensus. \( \Box \)

Lemma 3. Any message \( m \) R-delivered at a correct process \( p_i \) and such that \( m.state = q_0 \) eventually progresses to \( m.state = q_1 \) (or, equivalently, \( \forall p_i \) that is correct and belongs to a group \( g_x \in m.dest \), the thread \( T_i^m \) does not remain permanently blocked at line 1.2).

Proof. Let us consider a group \( g_x \in m.dest \). Due to the Termination property of the underlying uniform reliable multicast, if \( \exists p_i \) such that \( m \in Rec_i \) with \( m.state = q_0 \) then, for any correct process \( p_j \) of \( m.dest \) (and, hence, for all correct processes of \( g_x \)), we will have \( m \in Rec_j \) with \( m.state = q_0 \).

The proof is by contradiction. Let us assume that \( m \) is never output as a result of a consensus executed by \( g_x \). The number of messages \( m' \) that precede \( m \) in a queue \( Rec_i \) and that are such that \( m'.state = q_0 \) or \( q_2 \) is finite. So, the total number of messages \( m' \) that precede \( m \) in any queue \( Rec_i \) (with \( p_i \in g_x \)) and that are such that \( m'.state = q_0 \) or \( q_2 \) is finite. Let \( M \) be this number.

Due to Lemma 2, noncrashed processes of \( g_x \) launch and execute the same sequence of consensus: \( \text{propose}(k_1, -) \),
Lemma 4. Any message \( m \) R-delivered at a correct process \( p_i \) and such that \( m:state = q_2 \) eventually progresses to \( m:state = q_3 \) (or, equivalently, \( \forall p_i \) that is correct and belongs to a group \( g_x \in m:dest \), the thread \( T_m \) does not remain permanently blocked at line 1.4).

Proof. Due to Lemma 3, a message progresses from \( q_0 \) to \( q_1 \) at all noncrashed processes of its destination groups. So, each noncrashed process of \( m:dest \) sends the timestamp \( m:ts_0 \) associated with \( m \) by its group \( g_x \). As there is at least one correct process per group \( g_x \) and as communications are reliable, it follows that a thread \( T_m \) (of a correct process \( p_i \in m:dest \)) cannot block forever at line 2.4. Consequently, any such thread \( T_m \) executes \( m:state = q_3 \).

Lemma 5. Any message \( m \) R-delivered at a correct process \( p_i \) and such that \( m:state = q_2 \) eventually progresses to \( m:state = q_3 \) (or, equivalently, \( \forall p_i \) that is correct and belongs to a group \( g_x \in m:dest \), the thread \( T_m \) does not remain permanently blocked at line 1.6).

Proof. Let us first note that, due to the previous lemma, each noncrashed process of \( g_x \) will have \( m \) in its queue Rec, with \( m:state = q_2 \). The proof is then by contradiction. It is similar to the one of Lemma 3. (Within \( g_x \), there is a finite number of consensus executions that can output a message \( m' \neq m \). So, \( m \) will eventually be output by a consensus and \( m:state \) will consequently be updated to \( q_3 \)).

Lemma 6. Let us consider two distinct messages \( m_1 \) and \( m_2 \) that have been timestamped. (Such a timestamp can be a timestamp proposed by a group \( g_x \) or a definitive timestamp.) These two timestamps are different.

Proof. If the timestamps have been forged by distinct groups, they differ in the field “group identity.” If they have been forged by the same group, due to line 2.3, they have different clock values.

4. Actually, within a group, for the consensus protocol to work, the group must contain a majority of correct processes (Section 3.3).

Lemma 7. Let us consider a message \( m \) that has been R-delivered by a process. Then, \( m \) will get a single definitive timestamp \( m:ts \) and \( m:ts \) will be greater than or equal to any of its group timestamps \( m:ts^k \) (\( g_x \in m:dest \)).

Proof. This lemma follows from Lemma 1, from the properties of uniform consensus (applied to consensus executed within each group belonging to \( m:dest \) and deciding \( m \) when \( m:state = q_0 \)), and from from lines 1.3-1.5.

Lemma 8. Any message \( m \) R-delivered at a correct process \( p_i \) and such that \( m:state = q_k \) will eventually be TO-delivered (or, equivalently, \( \forall p_i \) that is correct and belongs to a group \( g_x \in m:dest \), the thread \( T_m \) cannot remain blocked forever at line 1.7). Moreover, \( p_i \) TO-delivers messages according to the order defined by their definitive timestamps.

Proof. Let \( t_m^i \) be the time (local to \( p_i \)) at which \( m:state \) takes the value \( q_k \). From \( t_m^i \) on, all the messages \( m' \) that are in \( Rec \) with \( m':state = q_k \) and all the messages that will be R-delivered to \( p_i \) will get a timestamp higher than \( m:ts \). This is due to the fact that the clock value in \( m:ts \) is \( \leq \ clock \) and to the fact that the next action on \( clock \) will necessarily be an increment (line 2.3).

5.3 Global Total Order

Theorem 1. If 1) a correct process TO-multicasts \( m \) or if 2) a process TO-delivers \( m \), then all correct processes that belong to a group of \( m:dest \) TO-deliver \( m \).

Proof. Due to Lemma 1, if 1) a correct process TO-multicasts \( m \) or if 2) a process TO-delivers \( m \), then any correct process \( p_i \) that belongs to a group of \( m:dest \) launches a thread \( T_m \). Due to Lemmas 3, 5, and 8, it follows that any correct process \( p_i \in g_x \) TO-delivers \( m \).

6 Reducing the Cost of the Protocol

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
The proposed protocol enjoys an interesting methodological property. In “extreme” configurations, it becomes equivalent to well-known protocols.

- Let us first consider the case where there is a single group. In that case, the proposed protocol implements a Total Order Broadcast (also called Atomic Broadcast) primitive. As suggested in Section 4.2 (see the remark after the explanation of Step 4), the protocol can be simplified and we obtain a variant of a consensus-based Atomic Broadcast protocol defined by Chandra and Toueg [5].
- Let us now consider the case where each process constitutes a group. In that case, the consensus among the processes of a group is no longer necessary. So, the thread $T_{cons}^k$ reduces to the part of the process $p_i$ that manages its Lamport clock. The protocol can be simplified accordingly and we get a Total Order Multicast protocol similar to the one proposed by Skeen (this protocol is described in [2]).

Actually, the proposed protocol can be seen as a combination of these two particular protocols that produces a consensus-based protocol solving the Total Order Multicast to Multiple Groups problem.

8 Conclusion

This paper has investigated a consensus-based approach to solving 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. Moreover, it has been shown that the consensus cost of the protocol can be reduced.

Acknowledgments

The authors would like to thank the anonymous referees whose insightful comments helped us improve the presentation of the paper. Udo Fritzke Jr. was supported in part by a grant from the CNPq/Brazil.

References
