Optimistic Atomic Multicast in One Communication Delay

July 29, 2011

Abstract

1 Introduction

Some multicast primitives have been devised in such a way that multicast groups needed to communicate only when they had messages to exchange. These multicast primitives are called *genuine*. We argue that it is possible, however, to devise a multicast primitive that, although not genuine, can make use of some knowledge given by the application to figure out which groups *can* communicate with each other. With such knowledge, although message exchanges take place even when there is no application message being transmitted between some two groups, such exchanges happen only when they are able to send to – or receive from – one another. The primitive that makes use of such property we call *quasi-genuine*.

2 System model and definitions

In this section we introduce the underlying system model and define two higher-level abstractions: consensus and atomic multicast. As we discuss later in the paper, atomic multicast builds on the consensus abstraction.

2.1 Processes and communication

We consider a system $\Pi = \{p_1, ..., p_n\}$ of processes which communicate through message passing and do not have access to a shared memory or a global clock. The system is asynchronous: messages may experience arbitrarily large (but finite) delays and there is no bound on relative process speeds. We assume the benign crash-stop failure model: processes may fail by crashing, but do not behave maliciously. A process that never crashes is *correct*; otherwise it is *faulty*. We define $\Gamma = \{g_1, ..., g_m\}$ as the set of process groups in the system. Groups are disjoint, non-empty, and satisfy $\bigcup_{g \in \Gamma} g = \Pi$. For each process $p \in \Pi$, *group*(p) identifies the group p belongs to. For the sake of simplicity, we abuse the notation by writing "$p \in \gamma$", instead of "$\exists g \in \gamma : p \in g$", where γ is any set of groups, such that $\gamma \subseteq \Gamma$. Hereafter, we assume that each group can solve consensus, a problem we define next.

Processes communicate using fifo reliable multicast, defined by the primitives fr-mcast(s, m) and fr-deliver(m), where s is a set of groups and m is a message. Reliable multicast guarantees that (i) for any process p and any message m, p fr-delivers m at most once, and only if $p \in s$ and m was previously fr-mcast *(uniform integrity)*; (ii) if a correct process p fr-mcasts m, then eventually all correct processes $q \in s$ fr-deliver m *(validity)*; (iii) if p fr-delivers m, then eventually all correct processes $q \in s$ fr-delivers m *(uniform agreement)*; and (iv) if p fr-mcasts m and then m', then no q fr-delivers m' without first fr-delivering m *(fifo order)*.

2.2 Consensus

An important part of this work relies on the use of consensus to ensure that processes agree upon which messages are delivered and in which order they are delivered. We consider instances of consensus solved within a group. Moreover, we distinguish multiple instances of consensus executed within the same group with unique natural numbers. Consensus is defined by the primitives $propose_g(k, v)$ and $decide_g(k, v)$, where g is a group, k a natural number and v a value, and satisfies the following properties in each instance k [**?**]: (i) if process $p \in g$ decides v, then v was previously proposed by some process in g *(uniform integrity)*;

(ii) if $p \in g$ decides v, then all correct processes in g eventually decide v *(uniform agreement)*; and (iii) every correct process in g eventually decides exactly one value *(termination)*.

2.3 Atomic multicast

Atomic multicast ensures that messages can be addressed to a set of groups. Atomic multicast is defined by the primitives multicast((m)) and deliver((m)), and guarantees the following properties.

(i) If a correct process p multicasts m, then every correct process $q \in m.dst$ delivers m *(uniform validity)*.

(ii) If p delivers m, then every correct process $q \in m.dst$ delivers m *(uniform agreement)*.

(iii) For any message m, every correct process $p \in m.dst$ delivers m at most once, and only if some process has multicast m previously *(uniform integrity)*.

(iv) If processes p and q are both in $m.dst$ and $m'.dst$, then p delivers m before m' if and only if q delivers m before m' *(atomic order)*; moreover, if p multicasts m and then m', then no process q in both $m.dst$ and $m'.dst$ delivers m' before delivering m *(fifo order)*.

Atomic multicast encompasses atomic broadcast. With atomic broadcast, every message is always multicast to all groups. Therefore, it is simple to implement atomic multicast using an atomic broadcast algorithm: To multicast message m, it suffices to broadcast m to all groups; those groups not included in $m.dst$ discard m while groups in $m.dst$ deliver m. Obviously, this algorithm defeats the purpose of atomic multicast, namely, performance. In order to rule out such bogus implementations, some early work introduced the notion of genuine atomic multicast algorithm [**?**].

Intuitively, a genuine atomic multicast protocol spares unnecessary communication among groups, that is, to deliver message m, groups g and g' only communicate if they are "concerned by m"—group x is concerned by m if the process that multicasts m is in x or $x \in m.dst$. While genuineness is an important property for atomic multicast protocols, it has been shown to be expensive. More precisely, no genuine multicast protocol can deliver messages in fewer than two network delays [**?**]. Since we seek communication-efficient algorithms, we introduce next the concept of quasi-genuine atomic multicast protocols.

For any group $g \in \Gamma$, we define $sendersTo(g)$ as the set of groups that can multicast a message to g. $sendersTo(g)$ is application specific and statically defined. In Section **??** we discuss how it can be dynamically defined. In a quasi-genuine multicast protocol, groups g and g' can communicate if $g \in sendersTo(g')$.

WE BADLY NEED AN EXAMPLE HERE...

3 Baseline atomic multicast

In this section, we describe a simpler, more easily understandable version of our proposed multicast protocol, upon which several optimizations are made and presented afterwards. Here, we focus on the main principles of the algorithm, which are: fifo delivery of messages, consensus within each group regarding the order of messages from that group and the use of barriers to synchronize the delivery of messages across different groups.

In Section 3.1, we give a high-level description of our baseline multicast protocol and, in Section 3.2, we present its algorithm and explain it in detail.

3.1 Overview of the algorithm

Hereafter, we assume that in addition to application data, each message m has four fields: its set of destination groups $m.dst$, its source group $m.src$, its timestamp $m.ts$ and an unique message identifier $m.id$. To multicast m, process p in group g sets $m.ts$ to a unique timestamp based on its real-time local clock and then fr-mcasts m to all processes in g.

When processes in g fr-deliver m, they run a consensus instance to agree on the final timestamp assigned to m, after possibly adjusting it to ensure the following invariant: for any two messages m and m', multicast by processes in g, if a process in g decides m before m', then $m.ts < m'.ts$. This is important since we intend to deliver messages according to their timestamp order.

Notice that the original timestamp assigned by the message's sender may violate the invariant due to the assynchrony of the system. In such a case, after consensus processes reassign the message's timestamp to make the condition hold.

Once group g has decided on m's final timestamp, m is fr-mcast to all its destination groups. Let h be a group in $m.dst$. To ensure that no message m', where $m'.ts < m.ts$, is delivered after m, process $q \in h$ will deliver m only when it has received at least one message from every group i in $sendersTo(h)$ with a timestamp greater than m's timestamp.

3.2 Detailed description

A more formal description of the protocol is given in Algorithm 1. We use a $getTime()$ primitive, which returns the current value of the local wallclock, which is monotonically increasing. We also use the concept of *barrier*. A barrier is the certainty that a given group will not send any new messages with a timestamp lower than a given value to some other group. As all groups receive messages from each other in the order of their timestamps, which is done by using fr-mcast, every message is seen as a barrier.

Moreover, each process p of group g keeps some sets of messages:

- $messages_g$, containing the messages that have been multicast by some process of group g;
- $decided_g$, which contains the messages that have already been proposed and decided within $group(p)$;
- *stamped*, containing the messages sent to g by any group; a final timestamp has already been assigned to each of these messages, although, in order to be delivered, some of them may still need barriers from the groups in $sendersTo(G)$;
- *delivered*, which contains the messages that have been already delivered by p.

Whenever p has some message m to multicast, it assigns to m an initial timestamp value, based on the local wallclock of p, after which m is fr-mcast to all the other processes of g (l. 6 and l. 7). After fr-delivering m, each process puts it in $messages_g$ to be proposed in g until it is decided (l. 16 and l. 17), along with other undecided messages. As all correct processes of g do so, some accepted proposal will end up containing m. Even though m might not have its source group as a destination, m is still agreed upon in its group of origin, so its order among other messages from g is decided and it may be retrievable even in the presence of failures.

When a set of messages is decided, such messages are handled by each process of g in ascending order of timestamps, as if such messages had been decided separately, in the original timestamp order. If m has a timestamp that is lower than the timestamp of some previously decided message m', the timestamp of m is changed to a value greater than $m'.ts$ (l. 20 to l. 23). Doing so will ensure that all messages from g are decided by p in the same order of their final timestamps.

After deciding m, p checks whether g is one of its destinations. If this is the case, m is inserted into *stamped* (l. 25), meaning that p should deliver it eventually. Then, m is inserted into $decided_g$, so that p knows that m was decided already and may stop proposing it in the following consensus instances. As no message decided afterwards within g will have a timestamp lower than that of m, p sets $barrier(g)$ to $m.ts$ (l. 27). Finally, if m has any destination other than g, p fr-mcasts m to it (l. 28).

Let h be any possible destination group of m other than g. When some process q of h fr-delivers m for the first time – m may be fr-delivered multiple times, since all correct processes of g fr-mcast m to h –, q sets $barrier(g)$ to $m.ts$ (l. 13), since, from the fr-mcast primitive, any previous message from g has already been received. Besides, q knows that the processes of g have already decided the final timestamp of m, so q inserts it into *stamped* (l. 12).

For any process r, of any group i in $m.dst$, when m belongs to *stamped*, having the lowest timestamp among all undelivered messages of such set (l. 30), and r has already received a barrier from every group in $sendersTo(i)$, where the value of such barrier is greater than $m.ts$ (l. 31), m is delivered by r (l. 32). Doing so, no message m' with a timestamp greater than $m.ts$ will be delivered before m by process r. If two messages m_1 and m_2 have the same timestamp, we break ties using their message identifier. More precisely, if $m_1.ts = m_2.ts \wedge m_1.id < m_2.id$, we consider that $m_1.ts < m_2.ts$.

Algorithm 1 multicast(m) – executed by process p in group g

```
1:  Initialization
2:      $k \leftarrow 0$, $messages_g \leftarrow \emptyset$, $decided_g \leftarrow \emptyset$, $stamped \leftarrow \emptyset$, $delivered \leftarrow \emptyset$
3:      for all $h \in sendersTo(g)$ do
4:          $barrier(h) \leftarrow 0$

5:  To multicast a message $m$
6:      $m.ts \leftarrow getTime()$
7:      fr-mcast($\{g\}$,$m$)

8:  when fr-deliver($m$)
9:      if $g = m.src$ then
10:         $messages_g \leftarrow messages_g \cup \{m\}$
11:     else if $m \notin stamped$ then
12:         $stamped \leftarrow stamped \cup \{m\}$
13:         $barrier(m.src) \leftarrow m.ts$

14: when $messages_g \setminus decided_g \neq \emptyset$
15:     $k \leftarrow k + 1$
16:     $undecided \leftarrow messages_g \setminus decided_g$
17:     propose$_g(k, undecided)$
18:     wait until decide$_g(k, msgSet)$
19:     while $msgSet \setminus decided_g \neq \emptyset$ do
20:         let $m$ be the message in $msgSet \setminus decided_g$ with smallest timestamp
21:         let $m'$ be the message in $decided_g$ with greatest timestamp
22:         if $m'$ exists and $m'.ts > m.ts$ then
23:             $m.ts \leftarrow m'.ts + 1$
24:         if $g \in m.dst$ then
25:             $stamped \leftarrow stamped \cup \{m\}$
26:         $decided_g \leftarrow decided_g \cup \{m\}$
27:         $barrier(g) \leftarrow m.ts$
28:         fr-mcast($m.dst \setminus \{g\}$,$m$)

29: when $stamped \setminus delivered \neq \emptyset$
30:     let $m$ be the message in $stamped \setminus delivered$ with smallest timestamp
31:     if $\forall h \in sendersTo(g) : m.ts < barrier(h)$ then
32:         deliver($m$)
33:         $delivered \leftarrow delivered \cup \{m\}$
```

3.3 Addressing liveness

Algorithm 1 does not guarantee liveness: a process $p \in g$ cannot deliver a message m until it has received some barrier b from every group in $sendersTo(g)$, where $b > m.ts$. The problem is that, if any group h in $sendersTo(g)$ has no more messages to send to g, p may not be able to deliver any more messages.

One way to ensure liveness is by sending empty messages from each group g to each group h, where $g \in sendersTo(h)$, when g has not sent any message to h for a while. This is a straight-forward approach to ensure progression; in Section 4.1, we present a more latency-efficient way to do it, based on sending empty messages when requested. The trade-off between those two approaches concerns message delivery latency against number of control messages exchanged. In either case, the empty messages are handled as ordinary ones, except they are never delivered to the application – calling deliver($null$), where $null$ is any empty message, has no effect.

Algorithm 2 describes the periodic sending of empty messages. Let h be any group, let g be any group in $sendersTo(h)$ and let p be some process of g. When p decides a message m, it knows that all correct processes of g have also decided m. As any other messaged decided within g afterwards will have a timestamp greater than $m.ts$, m serves as a barrier from g to itself. Besides, as m is decided by all correct processes of g, m will be fr-mcast and eventually received by its destinations, serving as a barrier from g to every group $i \in m.dst$ (l. 4 to l. 6 of Algorithm 2).

However, when there is a long period after the last time when some message has been sent from g to h, p decides to create some empty message to send to the processes of h with the sole purpose of increasing their barrier values and allow for the delivery of possibly blocked messages in that group. The value of $barrierThreshold$ defines how long a process should wait before creating an empty message. To guarantee that the empty message will be decided in g and fr-mcast to h, all processes of g create it and insert it into $messages_g$ to be proposed (l. 7 to l. 12).

Algorithm 2 Achieving liveness by sending periodic messages; executed by every process p of group g

```
1:  Initialization
2:    for all $h : g \in sendersTo(h)$ do
3:      $lastBarrierCreated(h) = 0$

4:  when inserting a message $m$ into $decided_g$
5:    for all $h \in m.dst \cup \{g\}$ do
6:      $lastBarrierCreated(h) \leftarrow m.ts$

7:  when $\exists h : g \in sendersTo(h) \wedge getTime() - lastBarrierCreated(h) > barrierThreshold$
8:    $null \leftarrow$ empty message
9:    $null.ts \leftarrow lastBarrierCreated(h) + barrierThreshold$
10:   $null.dst \leftarrow \{h\}$
11:   $messages_g \leftarrow messages_g \cup \{null\}$
12:   $lastBarrierCreated(h) \leftarrow null.ts$
```

4 Optimized atomic multicast

In Section 3, we presented a straight-forward version of our atomic multicast protocol, which, despite being easily understandable, is open to several optimizations, which are described hereafter. In Section 4.1, we present a more latency-efficient approach to ensure that processes will eventually receive all barriers necessary to deliver each message; in Section 4.2, we show that different instances of consensus may be voted upon and decided in parallel, further reducing the time needed for consensus; in Section 4.3, we give more details regarding how consensus is achieved within only three communication steps and, in Section 4.4, we present our optimistic delivery algorithm, which may deliver messages in one single communication step. Furthermore, in Section **??**, we present a better way to define the timestamps of the messages and, in Section **??**, we show how to relax the uniform validity property in order to reduce the final delivery time and the number of mistakes made by the optimistic delivery.

4.1 Requesting barriers

The problem with addressing liveness by sending empty messages periodically, as in Algorithm 2, is that a process q from some group h might have received some message m right after receiving a barrier b from some other group $g \in sendersTo(h)$, such that $b < m.ts$. This would mean that, if g has no messages to send to h, q will have to wait for, at least, $barrierThreshold$ – maybe just to receive from g some barrier $c : c < m.ts$, having to wait again and so on. How long exactly it will take for q to finally deliver m depends on several variables: besides the value of $barrierThreshold$, it depends on how far in the past m was created and how long it takes for some barrier $d : d > m.ts$ to arrive at q.

There is a way to provide liveness with a lower delay for delivering messages, although that would imply exchanging more messages. Let $blockers(m)$ be the set of groups whose barrier is needed in order for a given group to deliver m, i.e., $blockers(m) = \{g_b \in \Gamma : \exists g_{dst} \in m.dst \land g_b \in sendersTo(g_{dst})\}$. The idea is that, once the processes of each group g_b in $blockers(m)$ have fr-mcast a barrier $b > m.ts$ to every group that might need it, i.e. to every group $g_{dst} \in m.dst : g_b \in sendersTo(g_{dst})$, all possible destinations of m can deliver it.

Instead of relying on periodic messages, as soon as a process p in a group g knows the final the final timestamp of a message m from g, p requests a barrier to every $q \in blockers(m)$. This request should be sent when the final timestamp of m is known because the barrier sent by q may be guaranteed to be greater than $m.ts$. Such request is sent to the processes in $blockers(m)$, so that they know that there is a message whose delivery will be blocked until they send a proper barrier to unblock it.

When the process q of some group h receives a barrier request for a message m, q knows that there are other groups depending on the barrier of h to deliver m. For that reason, q will create an empty message $null$ with a timestamp greater than $m.ts$ and its destination groups will be those in $m.dst$ that can receive messages from h, i.e., $null.dst = m.dst \cap \{i \in \Gamma : h \in sendersTo(i)\}$. Once $null$ is proposed and decided in h, each process of h will fr-mcast it to every group in $null.dst$. This way, any group which was waiting for a barrier from h to deliver m will receive the message $null$.

4.2 Parallel instances of consensus

We can further optimize the delivery algorithm. One major problem with Algorithm 1 is that the processes wait until a consensus instance has been finished to start a new one, so that no message is proposed twice. However, even if a message is proposed and accepted twice, each process may choose to consider only the first time (i.e. the consensus instance with lowest id) when a message m has been proposed. This would require some more processing and could cause some unnecessary traffic due to messages being proposed in different consensus instances, but it would reduce the average time needed to achieve consensus.

The basic idea would be that, whenever a process p from group g has an undecided message m in $messages_g$, p will propose m in some instance k as soon as m is fr-delivered. As p cannot foresee whether m will be decided in k or not, it keeps the double (m, k) in a $trying$ set. When k terminates, (m, k) is removed from $trying$ and p checks whether m has been decided – which might have happened also in some instance $k' \neq k$ in the meantime – by checking its $decided_g$ set. If not, p proposes m again in some consensus instance k'' and inserts (m, k'') into $trying$. If a message m is agreed upon in some consensus instance k, the decision of m in any instance $k' : k' > k$ is ignored.

As processes do not block while running a consensus instance, there is no guarantee that the consensus instances will terminate in the correct order, which is: when a process p from group g decides (k, val), p has already decided (k', val') for any instance $k' < k$, unless k was the first consensus instance within g. However, we assume that this order is followed, which can be provided by the consensus implementaion by simply using a sequence number. This way, we can abstract the details of decision ordering.

4.3 Using Paxos with a leader for consensus

Although ensuring termination of consensus in an asynchronous system is not possible [?], the Paxos algorithm [?] can guarantee the termination of a consensus instance k as long as some assumptions are held:

- the maximum message delay bound δ is known;

- at some point in the execution of k, there is a leader which does not fail until a value has been chosen[1];
- during the whole execution of k, more than half of the processes are correct, that is, if the number of faulty processes is f and the total number of processes is n, then $f < \lceil \frac{n}{2} \rceil$;
- enough messages are successfully received, that is, for each phase of the execution of k, a majority of processes succesfully receive the leader's message or the leader successfully receives the reply (be it a confirmation or a negative acknowledgement) from a majority of processes.

Paxos uses the abstraction of *proposers*, *acceptors* and *learners*. In short, the acceptors are the processes which have to agree upon some value given by the proposers. The learners are those who are notified about which value has been accepted. With a leader, Paxos can achieve consensus in most instances, except the first one since the last leader change, in 3δ – the tree phases are: a proposal is sent to the leader, which forwards it to the other processes (acceptors) and receives a confirmation. Each acceptor can send the confirmation to every learner, so that each learner can figure out by itself that a value has been agreed upon once it receives a confirmation from a majority of acceptors.

In the context of our multicast primitive, we consider that every correct process of a group is proposer, acceptor and learner in every consensus instance run by that group. Besides, we assume that one of such processes has been elected as leader. The first phase of Paxos – forwarding a proposal to the leader – is already done when the source process p of a message m fr-mcasts it to all process in $group(p)$. The last phase message can also be sent to all destination processes of m, so that they can infer that m has been accepted in its source group. For the consensus instance deciding each message m within a given group, each process in $m.dst$ would also be a learner for that instance. This would be done instead of the processes of $group(p)$ waiting for m to be decided to only then fr-mcast it of to all its destinations. By doing this, every message m could be decided and sent to every destination process within 3δ from when m was created.

One problem could be that, if timestamps are defined as in Algorithm 1, each destination process would need to know the contents of $decided_g$ to infer the final timestamp of any message m sent by group g. However, we are assuming that there is a leader process, by which all proposals have to pass. Such leader can, therefore, predict what will be contained in $decided_g$ and set the final timestamp of m before sending it to the acceptors in g in the second phase, so that the confirmation message from the acceptors in the last phase of Paxos would already contain the final timestamp of m. Besides, the leader could request the barriers necessary for the delivery of m at the beginning of the second phase of Paxos.

Figure **??** illustrates the execution: process p_1 from group g, whose leader is p_3, multicasts m. In the first phase of the algorithm, m is fr-mcast to p_3. Then, in the second phase, p_3 has to change $m.ts$ because some other message with higher timestamp has passed by p_3 previously. Then, still in the second phase, p_3 sends m, with its new timestamp, to all acceptors, i.e., all processes of g. Finally, in the last phase, the processes in h receive the confirmation message from the acceptors of g, already containing the final timestamp of m.

4.4 Optimistic Delivery

Even if a fifo reliable multicast primitive is used to send every message to each one of its destinations, guaranteeing its arrival in fifo order, messages from different senders may arrive in different orders at different destinations, so consensus is necessary to decide which order should be considered by all processes. However, we can predict the final delivery order using the timestamps assigned to the messages by their senders: if each process p waits long enough before proposing some message m, every message $m' : m'.ts < m.ts$ will arrive eventually and p will be able to propose them in the same order of their initial timestamps, achieving total order without needing any consensus for that.

The problem is then how to define the length of such wait window for each message m such that every message prior to m will have already been received when the window time has elapsed. As the message delay is unpredictable in asynchronous systems, we make an optimistic assumption:

A2: every process p knows a value $w(p)$, which is at least the maximum sum of the message delay bound plus the clock deviation between p and any process p' which could send a message to p.

[1] Lamport demonstrates in [?] that, if the time needed to elect a leader is T_{el}, the time needed to conclude a consensus instance would be at most $T_{el} + 9\delta$ after the last leader failure during the execution of k.