Optimistic Atomic Multicast in One Communication Delay

July 27, 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 [2]: (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. 12), 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 ← 0, messages_g ← ∅, decided_g ← ∅, stamped ← ∅, delivered ← ∅
3:    for all h ∈ sendersTo(g) do
4:      barrier(h) ← 0

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

8:  when fr-deliver(m)
9:    if g = m.src then
10:     messages_g ← messages_g ∪ {m}
11:   else if m ∉ stamped then
12:     stamped ← stamped ∪ {m}
13:     barrier(m.src) ← m.ts

14: when messages_g \ decided_g ≠ ∅
15:   k ← k + 1
16:   undecided ← messages_g \ decided_g
17:   propose_g(k, undecided)
18:   wait until decide_g(k, msgSet)
19:   while msgSet \ decided_g ≠ ∅ do
20:     let m be the message in msgSet \ 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 ← m'.ts + 1
24:     if g ∈ m.dst then
25:       stamped ← stamped ∪ {m}
26:     decided_g ← decided_g ∪ {m}
27:     barrier(g) ← m.ts
28:     fr-mcast(m.dst \ {g},m)

29: when stamped \ delivered ≠ ∅
30:   let m be the message in stamped \ delivered with smallest timestamp
31:   if ∀h ∈ sendersTo(g) : m.ts < barrier(h) then
32:     deliver(m)
33:     delivered ← delivered ∪ {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 a 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. We describe here how liveness can be ensured by sending periodic empty messages from each group g to each group h, where $g \in sendersTo(h)$, to which no message has been sent for a specified time period.

This is a quite 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 sent. In either case, the empty messages are handled as ordinary messages, 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. The variable *barrierThreshold* defines how long a process should wait before sending an empty message to some group. When process $p \in g$ decides a message m, p knows that the other processes of g have also decided m, serving 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 $h : g \in sendersTo(h)$ (l. 12 of Algorithm 1). However, when there is a long period after the last time when such kind of message has been created, 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 h.

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

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

4:  when inserting a message m into decided_g
5:    for all h ∈ m.dst ∪ {g} do
6:      lastBarrierCreated(h) ← m.ts

7:  when ∃h : g ∈ sendersTo(h) ∧ getTime() − lastBarrierCreated(h) > barrierThreshold
8:    null ← empty message
9:    null.ts ← lastBarrierCreated(h) + barrierThreshold
10:   null.dst ← {h}
11:   messages_g ← messages_g ∪ {null}
12:   lastBarrierCreated(h) ← null.ts          {saving that nothing was sent until null.ts}
```

The problem with addressing liveness this way is that a process q from group h might have received some message m and has also just received a barrier b from $g \in sendersTo(h)$, 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 dependon 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.

It is necessary to guarantee that, whenever the *barrierThreshold* time has elapsed since the last message to a given group h has been created, an empty message will eventually be decided and sent to g, so that progression in g is guaranteed. Therefore, such empty message is created and proposed by every process in each group g: if some process of g does not propose the empty message, it might never be decided in such group.

4 Optimized atomic multicast

4.1 Requesting empty messages

There is a way to provide liveness with a lower delay for delivering messages, although that would imply creating more messages and making deeper changes in the delivery algorithm. Let $blockers(m)$ be defined as the set of groups whose barrier is needed in order for some group to deliver m. More formally, $blockers(m) = \{G_B \neq m.src : \exists G_{dst} \in m.dst \wedge G_B \in sendersTo(G_{dst})\}$. The idea is that, once each group G_B in $blockers(m)$ has sent[1] a barrier $b > m.ts$ to all the groups belonging to $m.dst \cap receiversFrom(G_B)$, all possible destinations of m can deliver it. This way, instead of relying on periodic messages, whenever a process p in a group G knows that a message m has just been decided in some consensus instance within G, p requests a barrier to every $p' \in blockers(m)$. This request is only sent after m has been decided, because the timestamp of m may have changed, which happens after the consensus. Such request is sent to the processes in the groups inside $blockers(m)$, so that they know that there is a message whose delivery will be blocked until they send a proper barrier to unblock it. Finally, a group never blocks the delivery of its own messages – hence the condition $G_B \neq m.src$ – because the message itself can be seen as a barrier from its source group.

When the process p' of some group G' receives a barrier request for a message m, p' knows that there are other groups depending on the barrier of G' to deliver m. For that reason, it will immediately create a *null* message with a timestamp equal to $m.ts$ and with $null.dst = m.dst \cap receiversFrom(G')$. Then, p' will insert such message in its $messages_g$ queue. Once the *null* message is proposed and decided in G', each process of G' will send it to every process in each group $G \in m.dst \cap receiversFrom(G')$. This way, any group which was waiting for a barrier from G' to deliver m will be able to do so as soon as it receives such *null* message. The new delivery algorithm would be as described in Algorithm 3.

4.1.1 Performance analysis

Assuming δ as the communication delay between every pair of processes, the time needed to decide a message m after it has been multicast by some process is equal, in the worst case, to $\delta + 2T_{cons}$, where δ is the time needed to fr-deliver a message and T_{cons} is the time needed to execute a consensus instance. This value is counted twice because m may have been inserted into $messages_g$ right after a consensus instance has been initiated. In that case, m would have to wait such consensus to finish, to be then proposed and finally decided. However, it may also happen that m has been inserted into $messages_g$ right before some proposal has been made, so it would take only $\delta + T_{cons}$ to decide m. As m may go into $messages_g$ anytime between the worst and the best case with the same probability, the average time needed for deciding m would be equal to $\delta + 1.5T_{cons}$. Nevertheless, the time needed to finally deliver m will depend on when barriers are received from other groups.

As discussed before, assuming δ as the communication delay between every pair or processes and T_{cons} as the time needed to run one consensus instance, we have a worst case time to decide a message within a group of $\delta + 2T_{cons}$. If there is any destination of m which is different from its source group, m is fr-mcast to it after its final timestamp has been defined, which takes another δ. At the same time (l. 35 of Alg. 3), m is fr-mcast also to the groups in $blockers(m)$, each of which then begins a consensus instance proposing an empty *null* message. After deciding *null*, it is fr-mcast to the processes of the groups which might be needing a barrier to deliver m. This way, we can calculate the number of communication steps needed to deliver a message, which is $3\delta + 4T_{cons}$ in the worst case and $3\delta + 2T_{cons}$ in the best case, leading to an average case of $3\delta + 3T_{cons}$.

Although we have bounded the delay to deliver a message, this came at the expense of creating a fairly high amount of control messages for requesting barriers. Unfortunately, this has to be done to guarantee that such *null* message will eventually be proposed, decided, and a barrier will be sent to some group which might be needing it.

4.2 Optimistic Delivery

Even if a reliable multicast primitive is used to send every message to each one of its destinations, guaranteeing its arrival, messages from different senders may arrive in different orders at different destinations, so consensus is necessary to decide which one should be considered by all processes. However, we can predict the final

[1]Here, by 'sending', we mean that the message has been received at the destination.

Algorithm 3 multicast(m) requesting empty messages – executed by every process p from group G

1: Initialization
2: $k \leftarrow 0$, $nextProp \leftarrow 0$, $decided_g \leftarrow \emptyset$, $delivered \leftarrow \emptyset$, $messages_g \leftarrow \emptyset$, $stamped \leftarrow \emptyset$
3: **for all** $G' \in sendersTo(G)$ **do**
4: $barrier(G', G) \leftarrow -\infty$

5: *To* multicast *a message* m
6: $m.ts \leftarrow (getTime(), 0)$
7: fr-mcast(m,$\{G\}$)

8: *When* fr-deliver(m')
9: **if** $G = m'.src$ **then**
10: $messages_g \leftarrow messages_g \cup \{m'\}$
11: **else if** $m' \notin stamped \wedge m' \notin delivered$ **then**
12: $barrier(m'.src, G) \leftarrow m'.ts$
13: **if** $G \in m'.dst$ **then**
14: $stamped \leftarrow stamped \cup \{m'\}$
15: **if** $G \in blockers(m') \wedge m' \neq null$ **then**
16: $null \leftarrow$ empty message
17: $null.src \leftarrow G$
18: $null.ts \leftarrow m'.ts$
19: $null.dst \leftarrow m'.dst \cap receiversFrom(G)$
20: $local\_msgs \leftarrow local\_msgs \cup \{null\}$

21: *When* $\exists m \in messages_g \wedge nextProp = k$
22: $messages_g \leftarrow messages_g \setminus decided_g$
23: **if** $messages_g \neq \emptyset$ **then**
24: $nextProp \leftarrow k + 1$
25: propose$_g$(k,$messages_g$)

26: *When* decide$_g$(k',$msgSet$)
27: **while** $\exists m \in msgSet : (\forall m' \in msgSet : m \neq m' \Rightarrow m.ts < m'.ts)$ **do**
28: $msgSet \leftarrow msgSet \setminus \{m\}$ {*the messages are handled in ascending order of timestamp*}
29: **if** $\exists m' \in decided_g : m'.ts \geq m.ts \wedge (\nexists m'' \in decided_g : m''.ts > m'.ts)$ **then**
30: $m.ts \leftarrow (m'.ts.rtc, m'.ts.seq + 1)$
31: **if** $G \in m.dst$ **then**
32: $stamped \leftarrow stamped \cup \{m\}$
33: $decided_g \leftarrow decided_g \cup \{m\}$
34: **if** $m \neq null$ **then**
35: fr-mcast(m,$(m.dst \setminus \{G\}) \cup blockers(m)$) {*the message is sent to blockers(m), as a barrier request*}
36: **else**
37: fr-mcast(m,$m.dst$) {*no need to unblock null, besides it would create infinite messages*}
38: $nextProp \leftarrow k' + 1$
39: $k \leftarrow k' + 1$

40: *When* $\exists m \in stamped : \forall G' \in sendersTo(G) : m.ts < barrier(G', G)$
$\wedge \nexists m' \in stamped : m'.ts < m.ts$
41: $stamped \leftarrow stamped \setminus \{m\}$
42: **if** $m \neq null$ **then**
43: deliver(m)
44: $delivered \leftarrow delivered \cup \{m\}$

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 it guarantees 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.

More formally, if $p \in G$, we can define $w(p)$ as $max_{p' \in sendersTo(G)}(\delta(p,p') + \epsilon(p,p'))$, where $\delta(p,p')$ is the maximum time a message takes to go all the way from p to p' and $\epsilon(p,p')$ is the difference between the clocks of p and p', so that clock deviations can be accounted for[2]. The clock deviation is mentioned in this assumption because the timestamp $m.ts$ of each message m is assigned by its sender according to its local clock and we want that, once a message m has been proposed, no message $m' : m'.ts < m.ts$ arrives afterwards. If our optimistic assumption considered only the message transmission delay, such property would not be guaranteed by it.

However, the assumption may not hold, so deliveries made based on it have to be confirmed. There would be then two deliveries for each message: an optimistic one, done within one communication step, and a conservative one, which may follow one of the algorithms described earlier. The difference is that, after a message m was fr-delivered at p, instead of proposing it, p inserts it into an *optPending* set, which is sorted in ascending order of the timestamps of the messages in it. Then, p waits until its own wallclock has a value greater than $m.ts.rtc + w(p)$, after which p both proposes m and opt-delivers (optimistically delivers) it. If *A2* holds, then all messages sent by $m.src$ and received after m has been opt-delivered will have a timestamp greater than $m.ts$.

The optimistic assumption allows for even further improvement. If it holds, the timestamp of the messages will never be changed (i.e. l. 30 of Alg. 3 will never be executed). Therefore, if Algorithm 3 is being used, there is no need to wait until a message m has been decided to know its final timestamp and only then request barriers to groups in $blockers(m)$. After the wait window for m has elapsed, such barrier request can already be sent.

If *A2* fails to hold and some message m is received by a process p after a message $m' : m'.ts > m.ts$ has been already opt-delivered and proposed by p, then the optimistic delivery algorithm made a mistake, which the application can figure out by the different delivery orders and take action to correct whatever problem this mistake might have caused. Besides, when such thing happens, it means that the timestamp of m has been changed by the conservative delivery algorithm and that, maybe, a new barrier request must be sent to $blockers(m)$ when this new timestamp is defined.

The delivery time of each message m will now depend on whether the assumption *A2* holds or not while m is being handled by the algorithm. If it does hold, it will be opt-delivered in the correct order within $w(p)$ and, as a barrier request was done in parallel with the conservative delivery algorithm, it will take $w(p) + 2T_{cons} + \delta$ to deliver m in the worst case. If it does not hold, after $w(p)$, m may be opt-delivered in an invalid order and, since its timestamp may have changed, a new barrier request may have to be done. Therefore, the worst case conservative message delivery time when the optimistic assumption does not hold and some message is delivered out of order would be $2w(p) + 4T_{cons} + \delta$.

Figure 1 illustrates both cases, when the optimistic assumption holds and otherwise. Three groups are shown: G, G' and G_b, where $sendersTo(G) = \{G_b\}$, $sendersTo(G') = \{G\}$ and $sendersTo(G_b) = \emptyset$. A message $m : m.src = G \wedge m.dst = \{G, G'\}$ is multicast and, at the same time, a barrier request is sent to the only group in $blockers(m)$, which is G_b. In (a), the optimistic assumption holds, so no message $m' : m'.ts < m.ts$ arrives after the proposal of m; in (b), however, it is necessary to make a second barrier request.

4.3 Parallel instances of consensus

We can further optimize the delivery algorithm. One major problem with proposing messages with consensus is that the previous algorithms wait until a consensus instance has been finished to start a new one, so that

[2]If this difference is less than zero, it means that the clock value in p' is higher than that in p, so p actually has to wait less time for messages from p', as they will have higher timestamps because of the clock deviation.

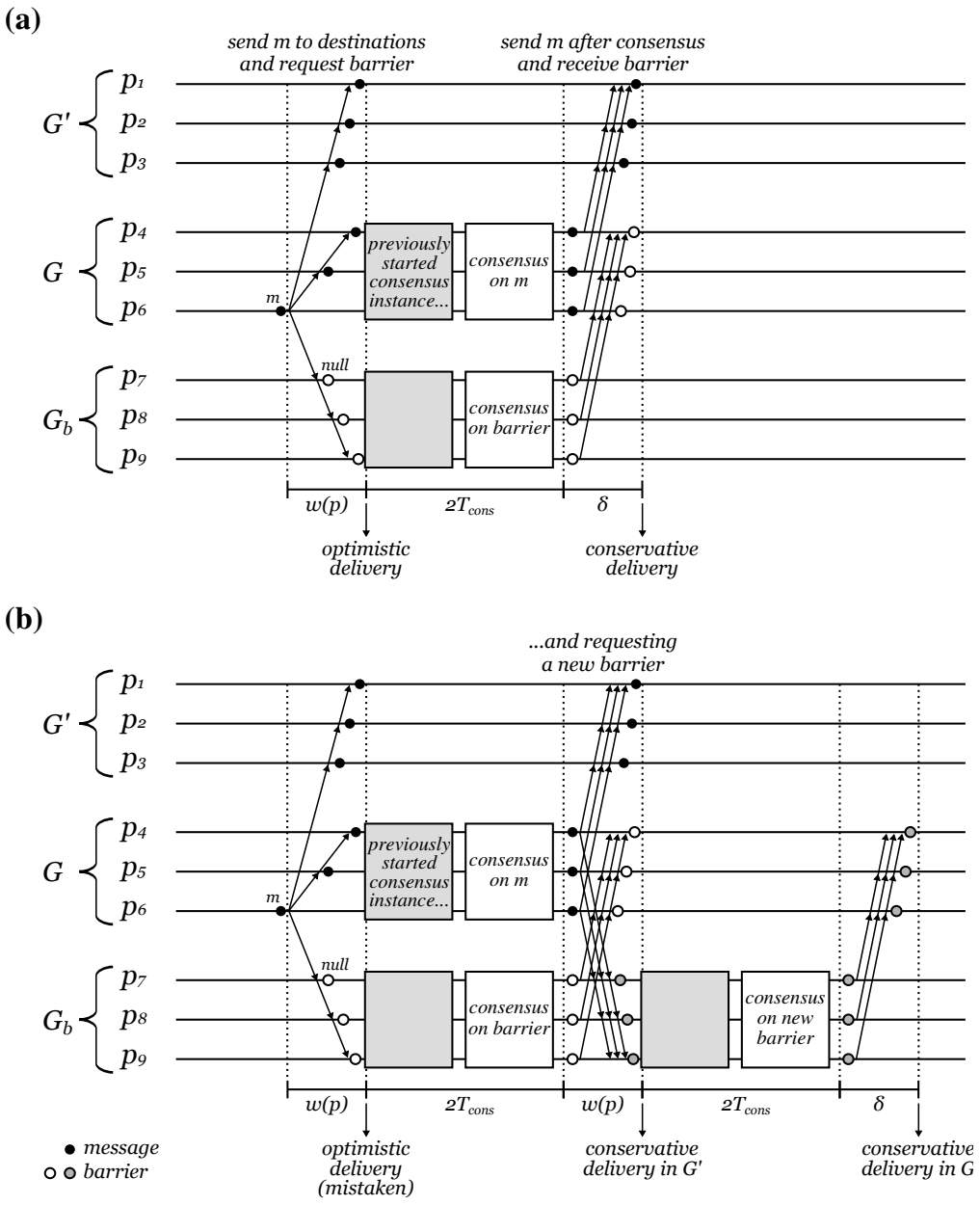

Figure 1: Execution examples when the optimistic assumption holds (a) and when it fails to hold (b)

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 local processing and could cause some unnecessary traffic due to messages being proposed in different consensus instances. The worst case delivery time of a message m, when using the optimistic delivery and barrier requests, would be $2w(p) + 2T_{cons} + \delta$. It must be noted, however, that this represents the case when the optimistic assumption fails to hold and m has its timestamp changed. Otherwise, the conservative delivery time would be $w(p) + T_{cons} + \delta$.

The basic idea would be that, whenever a process p has a pending message m that it received, it will propose it in some instance I as soon as its clock has a value greater than $m.ts.rtc + w(p)$. As p cannot foresee whether m will be decided in I or not, it keeps the double $(m, I.id)$ in a *trying* set. When I terminates, $(m, I.id)$ is removed from *trying* and p checks whether m has been decided – which might have happened also in some instance $I' \neq I$ in the meantime – by checking its $decided_g$ set. If not, p proposes m again in some consensus instance I'' and inserts $(m, I''.id)$ into *trying*.

Even if the consensus instances are run in parallel, we consider that they make a callback to $decide_g()$ in ascending order of instance id. This can be easily done by the consensus implementation using a sequence number. Whenever a message m is agreed upon in some consensus instance I, any later decision of m in some instance $I' : I'.id > I.id$ is merely ignored.

4.4 Using Paxos with a leader for consensus

Although ensuring termination of consensus in an asynchronous system is not possible [1], the Paxos [3] algorithm can guarantee the termination of an instance I as long as some assumptions are held:

- the maximum message delay bound δ is known;
- at some point in the execution of I, there is a leader which does not fail until a value has been chosen[3];
- during the whole execution of I, 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, $f < \lceil \frac{n}{2} \rceil$;
- enough messages are successfully received, that is, for each phase of the execution of I, 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 – forwarding a proposal to the leader, sending it to the other processes (acceptors) and receiving 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, the first communication step of Paxos – forwarding a message m to the leader – is already done by means of fr-mcasting each message to a set of processes that includes every process in m's group of origin. The last phase, which consists of every learner receiving the confirmation of a majority of acceptors, can also include the δ contained in the conservative message delivery time we analysed before: instead of waiting for a message to be decided and only then notifying other groups, the acceptors of a group can also send the confirmation to processes in other groups, so that also they can infer the first group's decision right away. This implies that, for each consensus instance run within a group G, each process in the groups of $receiversFrom(G)$ would be a learner for that instance. Figure 2 illustrates this: process p_1 from group G, whose leader is p_3, multicasts m; in the last phase of the consensus protocol, the processes in $G' \in receiversFrom(G)$ receive the confirmation message from the acceptors of G, so no extra communication step is needed to notify them about the final timestamp of m.

As m was fr-mcast in the beginning, there is no need to send all of its contents again in the consensus instance. Instead, only the timestamp of m is proposed, since this is the information that must be agreed upon – and which may change if the optimistic assumption does not hold and some other message of higher

[3]Lamport demonstrates in [3] 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 I.

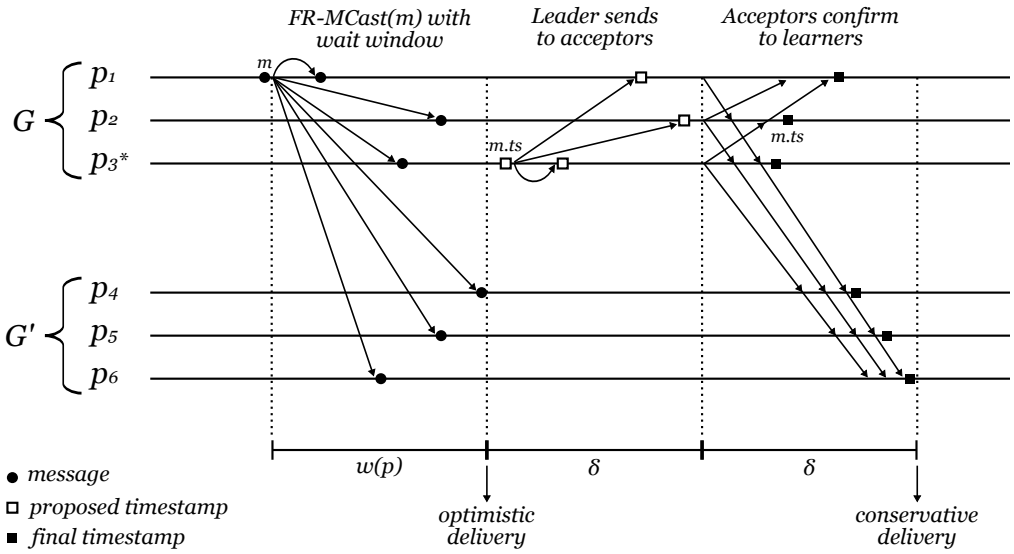

Figure 2: Deciding the final timestamp and notifying other groups in three communication steps

timestamp is delivered before m. In any case, as all messages agreed upon in G are notified to G', also the processes of G' are able to infer the final timestamp of each message from G.

As we are making an optimistic assumption – that each p waits for a time $w(p)$ long enough so that no timestamps must be changed – for the message delivery, the barriers are being requested to other groups in parallel, since m is also fr-mcast to $blockers(m)$ in the first communication step of our protocol. If such assumption holds, the barriers received will have a greater timestamp than m and, thus, the conservative delivery of m will take place $w(p) + 2\delta$ after it has been sent. If the clocks are perfectly synchronized and $w(p)$ does correspond to the maximum interprocess communication delay δ, the time needed to conservatively deliver a message using this algorithm would be 3δ.

Finally, we can guarantee a conservative delivery time of 3δ, if no message had its timestamp changed, because the barrier requests for that message were made based on its initial timestamp. If the timestamp has changed, then a new barrier request may have to be made to ensure the delivery of that message. Unfortunately, to guarantee the delivery of the message, such request can be done only after the final timestamp has been decided: once the source group of a message m noticed a change in $m.ts$, it sends another barrier request to $blockers(m)$. This request is handled in a way very similar to a conservative delivery, in the sense that, once it is received by a process, a null message is proposed, decided and then sent back to the destinations of m. This would take extra 3δ, so the worst case conservative delivery time would be 6δ.

Note, however, that for this worst case scenario to happen it would be necessary for the leader p of group G to: (1) not know some value greater than or equal to $w(p)$; (2) have received and proposed some message $m' : m'.ts > m.ts$ before m and (3) not have received any barrier with value greater than the new timestamp of m. Even after changing the timestamp of m, it may be not necessary to request a new barrier from each group in $sendersTo(G)$, because some other barrier – possibly requested because of some other message – with a value higher than the new timestamp of m may have been received already.

4.5 Relaxing validity and discarding stale messages

Although unlikely to happen, the worst case delivery time of multicast when using Paxos with a leader for consensus and requesting barriers in the beginning of the protocol is 6δ. It is so because some messages may need to have their initial timestamp changed as a possible consequence of the optimistic assumption *A2* not holding. However, a simple way to eliminate the possibility of having to change timestamps is by not considering messages whose delivery order does not follow the order of the initial timestamps.

Although discarding messages which do not follow the initial timestamp order conflicts with the properties we have defined for the multicast primitive, we describe it here because then every message would be delivered within 3δ, in the worst case. The validity property would now be changed to:

Optimistic Uniform Validity: if a process multicasts m and the optimistic assumption *A2* holds, then one of the correct processes that is a destination of m eventually delivers m.

The problem with changing the original uniform validity property is that it could make the multicast primitive impracticable for some applications. Nevertheless, some other applications not only do not require messages to be delivered if they arrive out of order, but also it is better not to deliver such messages at all – such as real-time streams of audio or video or online real-time multiplayer games.

5 Proof of correctness

Here, we prove that the multicast primitive ensures the properties defined in section 2 – uniform validity, uniform agreement, uniform integrity, uniform total order and fifo order. To do this, we prove that Algorithm 1, when used along with Algorithm 2, ensures these properties. Although there are different processes executing such algorithms at the same time, we assume that there is no concurrency on the execution of the algorithms in any single process.

5.1 Uniform Validity

Lemma 1. *Once every correct process in a group g has inserted a message m into $messages_g$, m will eventually be decided by all correct processes of g.*

Proof. In Algorithm 1, once each correct process p from g has received m and inserted into its $messages_g$ set, m will be proposed by p in every consensus instance, until m has been inserted into $decided_g$ (lines 16 and 17). As m is inserted into $decided_g$ only after being decided (l. 26), every correct processes of g will propose m at some point, so m will eventually be decided. From the uniform agreement property of consensus, as m is decided, all correct processes of g decide m. □

Lemma 2. *Once a message m has been multicast, it will be inserted into the stamped set of all the correct processes to which such message has been addressed.*

Proof. In Algorithm 1, whenever a process in a group g multicasts a message, it is first fr-mcast to all processes in g (l. 7). From the properties of the fr-mcast primitive, every correct process from g fr-delivers m and inserts it into $messages_g$ (l. 10). From Lemma 1, m will eventually be decided within g. When this happens, if $g \in m.dst$, every correct process of g also inserts m into *stamped* (l. 25). Besides, each correct process in g fr-mcasts m to all other groups that m is adressed to (l. 28).

Let q be any correct process in a destination group h of the message m, such that $h \neq g$. From the properties of fr-mcast, q will fr-deliver m. Once m is fr-delivered by q, such message is inserted into the *stamped* set of q, unless this has been already done (lines 11 and 12). □

Lemma 3. *Given a correct process p and a message m, p eventually receives some message $m' : m'.ts > m.ts$ from every group in $sendersTo(group u(p))$.*

Proof. Let $g = group(p)$ and let h be any group in $sendersTo(g)$. Every process $q \in h$ executes Algorithm 2, inserting messages with ever-increasing timestamp values into $messages_g$. Therefore, at some point, every process from h will propose some message $m' : m'.ts > m.ts$. As one of these proposals will be decided, then one of these messages will be fr-mcast to g. From the properties of fr-mcast, g will deliver it. □

Lemma 4. *Given any two messages m and m' sent from the same group g, if a process $p \in g$ decides that their final timestamps are such that $m.ts < m'.ts$, then p does not fr-mcast m' before m.*

Proof. Assume, by way of contradiction, that p fr-mcasts m' first. This means that m' was inserted into $decided_g$ first. When m is handled in the algorithm (lines 20 to 28), m' is already there. Then, p decides that the final timestamp of m is $m.ts > m'.ts$ (lines 20 to 23), which is a contradiction. □

Lemma 5. *Given any two correct processes and a message m, if each of such processes is either a destination of m or belongs to $m.src$, then such processes agree on the final timestamp of m.*

Proof. Let p, from group g, be the process that is multicasting m. We can prove this lemma by demonstrating that any other process $q \in g \cup m.dst$ will agree with p on the timestamp of m. Let us first consider the case where q also belongs to g. The proof for this case can be done by induction on the identifier k of each consensus instance within g:

Base case ($k = 1$): As this is the first consensus instance within g, it means that $decided_g$ was empty when such instance started. From the agreement property of consensus, we know that p and q decide the same contents for $msgSet$ (l. 18). Each message included in such agreed set also includes an initial timestamp field. As the $decided_g$ set is empty, such timestamps are not changed in neither p or q (lines 20 to 23). Therefore, both processes consider the same timestamp value for every message. Besides, as the $msgSet$ is the same for both, the $decided_g$ set remains identical in p and q.

Induction step: Suppose that p and q have already learnt the decisions of instance k, their $decided_g$ sets remained identical and they decided the same timestamp value for each message sent from some process in their group so far. From the algorithm, all processes within a group learn all decisions in the same order, which is that of the consensus instance identifiers. Therefore, both p and q will next learn the decision of the instance $k + 1$. From the agreement property of consensus, the $msgSet$ decided is the same for both processes. In the Algorithm 1, the timestamps may be changed after the consensus decision only in line 23, based on what is already in the $decided_g$ set (lines 20 to 23). As the $msgSet$ and $decided_g$ sets are each identical in p and q, they will make the exact same change to the timestamp of each message in the *while* loop (lines 20 to 28). Therefore, they also agree on the timestamps of each message decided on consensus instance $k + 1$ and their $decided_g$ sets remain identical.

Now, regarding the case of q not belonging to g, let r be any correct process of such group. After setting the final timestamp of m (l. 20 to l. 28), r fr-mcasts m to $group(q)$ (l. 28). From the algorithm, after fr-delivering m, q never changes the value of $m.ts$, set by q in accordance with p. □

Lemma 6. *If a process p from a group g has received a message m from another group $h \neq g$, then p has also received from h any message $m' : m'.ts < m.ts$.*

Proof. In Algorithm 1, every process q from h sends messages to the processes of g, including p, using a fifo channel, via the fr-mcast primitive. From the properties of consensus, we know that all correct processes from h will decide the same messages in each consensus instance k. From Lemma 4 and Lemma 5, every correct process from h fr-mcasts to g every message $m : m.src = h \wedge g \in m.dst$ (l. 28 of Algorithm 1) in the same order, which is that of the timestamps. From the fifo property of fr-mcast, we know that p will fr-deliver all of such messages from each process of h in ascending order of timestamps. Although such messages will arrive from different senders and may be out of order at p, once p received some message m from some process of h, p has also received any message $m' : m'.src = h \wedge g \in m.dst \wedge m'.ts < m.ts$. □

Lemma 7. *Once a process p has inserted a message m into its stamped set, no message $m' : m'.ts < m.ts$ from $m.src$ will be inserted into such stamped set afterwards.*

Proof. Let g be the group of p. Group g is either $m.src$ or not. In the former case, before inserting any message $m' : m'.src = m.src$ into *stamped*, p has to decide m' first. When a message is decided by p, p checks whether some other message which was decided previously has a timestamp lower than m' and, if that is the case, the timestamp of m' is changed to a value greater than that of any other message already decided (lines 20 to 23 of Algorithm 1). Only then m' may be inserted into the *stamped* set of p (l. 25). Therefore, p inserts messages from other processes of g into its *stamped* set in ascending order of timestamps. Besides, the messages from g are fr-mcast to other groups in this same order (l. 28) by all correct processes of g, from the uniform agreement property of consensus.

In the case where $g \neq m.src$, p inserts m into *stamped* in line 12. Considering that when p receives any message from $m.src$, any other other message from $m.src$ with a lower timestamp has already been received (from Lemma 6), we can infer that, once p inserts m into *stamped*, no message $m' : m'.ts < m.ts$ from $m.src$ will be inserted into the the *stamped* set of p afterwards. □

Lemma 8. *Once a message $m \neq null$ has been inserted into the stamped set of a correct process p, m is eventually delivered by p.*

Proof. Let $g = group(p)$. Eventually, p will have received, from every group in $sendersTo(g)$, some message with a timestamp greater than $m.ts$ (from Lemma 3). Once this happens, no more messages with a timestamp lower than that of m will be inserted into the *stamped* set of p (from Lemma 7 and the definition of $sendersTo(g)$). Let *ready* be the set of m plus all undelivered messages from *stamped* whose timestamps are lower than that of m, that is, $ready = \{m\} \cup \{m' : m' \in stamped \setminus delivered \wedge m'.ts < m.ts\}$.

We can infer that no more messages will be included in this set, meaning that any other message that might be later inserted into *stamped* will have a timestamp greater than that of any message in *ready*. Besides, the barriers received apply to all messages in this set, since their timestamps are not greater than $m.ts$. Each one of these messages will eventually be inserted into *delivered* and, if it is different from *null*, it will be delivered by p. We prove this by induction on the position i of each message m_i in *ready*, in ascending order of timestamps.

Base case ($i = 1$): Let m_1 be the first message in *ready*, i.e., $\nexists m \in stamped \setminus delivered : m.ts < m_1.ts$. We know that all the necessary barriers have been received already, so m_1 satisfies all conditions from lines 30 and 31 of Algorithm 1. Therefore, m_1 will be inserted into *delivered*, after which m_1 will no longer belong to $stamped \setminus delivered$. Also, if $m_1 \neq null$, m_1 will be delivered.

Induction step: Suppose that m_i will eventually be inserted into the *delivered* set. Once this happens, it means that m_i was the first message in $stamped \setminus delivered$ in ascending timestamp order. Since no more messages have been inserted into *ready*, as soon as m_i is inserted into *delivered*, m_{i+1} will be the first one in $stamped \setminus delivered$, having, therefore, the lowest timestamp in such set. Then, as all barriers necessary for m have already been received and $m_{i+1}.ts \leq m.ts$, m will satisfy both conditions of lines 30 and 31 of Algorithm 1. Thus, m_{i+1} will also be inserted into the *delivered* set and, if $m_{i+1} \neq null$, it will be delivered. □

Proposition 1. *Once a process multicasts a message, then all correct processes that are destinations of that message eventually deliver it.*

Proof. Immediate from Lemma 2 and Lemma 8. □

5.2 Uniform Integrity

Proposition 2. *If a message m was delivered by some process p of group g, then (1) m has been multicast before, (2) $g \in m.dst$ and (3) it has not been delivered by p before.*

Proof. From lines 30 to 33 of Algorithm 1, and since no message is removed from *delivered*, no message can be delivered twice, satisfying (3). For a message m to be delivered (l. 32), it must belong to the *stamped* set (l. 30). There are two possibilities to when m has been inserted into such set:

- m has been originated in the same same group of p, that is, $p \in m.src$, which means that m was inserted into *stamped* in line 25 of Algorithm 1, or
- m has been sent from a group $h \neq g$, which means that it was inserted by p into *stamped* in line 12.

For the case when $p \in m.src$, m can be inserted into *stamped* only in line 25, which means that $g \in m.dst$, satisfying condition (2) for this case. Also, this happens only when m has been decided in some consensus instance within g. To have been decided within g, from the properties of consensus, we know that it must have been proposed by some process of g, which happens in line 17. In line 17, for a message to be proposed, it must be in $messages_g \setminus decided_g$, as the contents of such set are the value proposed. For a message to be in $messages_g$, it must have been inserted there, which happens only in line 10. For l. 10 to be executed, m must have been fr-delivered and g must be the source group of m, that is, $g = m.src$. For a message to be fr-mcast by a process to its own group, a multicast(m) call must have been made – which satisfies condition (1) –, for line 7 is the only one where a process fr-mcasts a message to its own group.

As for the case when $p \notin m.src$, m is inserted into the *stamped* set of process p in line 12, when it has been fr-delivered. For a process $q \in h$, where $h = m.src$, to fr-mcast m, m must have been decided within h, which means that it was proposed within h. Therefore it was multicast – satisfying condition (1) – by h to g. From line 28, g necessarily belongs to $m.dst$, satisfying condition (2).

Finally, as for Algorithm 2, *null* messages are never delivered, so they will never violate the uniform integrity property. □

5.3 Uniform Agreement

Proposition 3. *If a correct process which is a destination of a message m delivers it, then all correct processes that are also destinations of m deliver it as well.*

Proof. Immediate from Proposition 1 and Proposition 2. □

5.4 Atomic Order

Lemma 9. *If two messages m and m', both different from null, have a correct process p as a destination, and $m.ts < m'.ts$, then p delivers m' after delivering m.*

Proof. Suppose, by way of contradiction, that m' is delivered by p before m. Message m either has already been inserted into *stamped* or not. In the former case, as $m \neq null$ and m has not been delivered, then m belongs to $stamped \setminus delivered$. Therefore, m' could not have been delivered, since $m.ts < m'.ts$ and it would not satisfy the condition from line 30 until m has been delivered, so we have a contradiction in the case where m was already in *stamped*.

The other case is when m has not been inserted into *stamped*. From line 31 of Algorithm 1, and since m' has been delivered, we know that p has received some barrier $b > m'.ts$ from every group in $sendersTo(group(p))$. Therefore, from Lemma 7, and since any barrier is also a message, we know that any new message that arrives at p will have a timestamp greater than $m'.ts$. As m has arrived after m', then $m.ts > m'.ts$, which is also a contradiction.

We have proven that the timestamp order will not be violated by p. Considering that, once a message has been multicast, it will be delivered (Proposition 1), then we know that all messages will be delivered by p, and in the correct timestamp order. □

Proposition 4. *If processes p and p' are both in $m.dst$ and $m'.dst$, then p delivers m before m' if, and only if, p' delivers m before m'.*

Proof. Immediate from Lemma 5, Lemma 9 and Proposition 3. □

5.5 Fifo Order

Lemma 10. *If a process p, from group g, multicasts m' after m, then the final values of their timestamps will be such that $m.ts < m'.ts$.*

Proof. As both messages are sent from g, they have to be decided withing g. If m and m' are decided in different consensus instances, and m is decided first, then, from lines 20 to 23, m' will necessarily have a timestamp greater than that of m. Therefore, the two possible ways of having $m'.ts < m.ts$ are: either m' is decided before m, or both are decided in the same consensus instance, but the timestamp of m is set to a value higher than that of m'.

From lines 6 and 7, and the properties of fr-mcast, we know that all correct processes of g fr-deliver m, then m', which means that each process q from g also inserts them into its $messages_g$ set in this same order. Suppose, by way of contradiction, that m' is decided before m. This implies that some process $q \in g$ proposed a message set that included m', but not m. As q inserted m before m' into $messages_g$, then m had already been inserted by q into $decided_g$. This means that m has been decided previously, which is a contradiction. Therefore, m' is not decided before m.

As for the case where m and m' are decided in the same consensus instance, we know, from line 6, that the initial timestamp of m is already smaller than that of m'. Therefore, from line 20, m is inserted into $decided_g$ first. Even if the timestamp of m has changed, as m was already in $decided_g$ before m', we know that the final timestamp of m' will again be made greater than that of m (from lines 20 to 23). □

Proposition 5. *If a process p multicasts m before multicasting m', then no process $q \in m.dst \cap m'.dst$ delivers m' before delivering m.*

Proof. Immediate from Lemma 9 and Lemma 10. □

6 Related work

[4]: optimistic total order bcast in wans: for the opt-delivery to work properly, requires that the delay between each pair of processes stay constant (ours only requires that it never goes beyond $w(p)$ for each process $p \ldots$). sequencer based (no tolerance for failures of the sequencer). not mentioning multicast.

7 Experimental results

8 Conclusion

References

[1] Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM 32* (April 1985), 374–382.

[2] Hadzilacos, V., and Toueg, S. Fault-tolerant broadcasts and related problems. In *Distributed systems (2nd Ed.)* (1993), ACM Press/Addison-Wesley Publishing Co., pp. 97–145.

[3] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems (TOCS) 16*, 2 (1998), 133–169.

[4] Sousa, A., Pereira, J., Moura, F., and Oliveira, R. Optimistic total order in wide area networks. In *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on* (2002), IEEE, pp. 190–199.