A Quasi-Genuine FIFO Total Order Multicast Primitive

July 28, 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

We assume a system composed of processes, regardless of how they are distributed – they may consist from local processes in the same device to nodes distributed over a large geographical area. Processes may fail by crashing and stoping, but they do not experience arbitrary behavior (i.e., no Byzantine failures). A correct process is operational "forever" and can reliably exchange messages with other correct processes. This assumption is only needed to prove liveness properties about the system. In practice, "forever" means long enough for one instance of *consensus* (defined below) to terminate. Every process p in the system belongs to one group, denoted by $group(p)$, which is a non-empty set of processes. We assume that consensus is solvable within each group.

Communication is done by message passing. Messages can be lost but not corrupted. If a message is repeatedly resubmitted to a correct process, it is eventually received. In particular, being m a message and gs a set of groups, we also use a *fifor-mcast*(m, gs) primitive, which is a FIFO reliable multicast primitive that sends message m to every process of each group in gs. If no failures occur, the delivery – *fifor-deliver* – of the messages sent with this primitive is done in FIFO order and in one communication step. By *communication step*, we mean the time needed for a message, after being sent by a process, to be delivered at another process. However, there are no bounds on this time, for actions to be executed or for the drift between the clocks of the different processes, which implies that the considered system is asynchronous.

2.1 Consensus

An important part of this work relies on the use of consensus to ensure that processes will agree upon which messages are delivered, and upon the order in which such messages are delivered. In the consensus problem, processes propose values and must reach agreement on the value decided. Uniform consensus is defined by the primitives *Propose(v)* and *Decide(v)* and satisfies the following properties [2], which we adapted to consider that each consensus is achieved within some group:

Uniform integrity: if process p decides v, then v was previously proposed by some process of $group(p)$.

Uniform agreement: if process p decides v, then all correct processes from $group(p)$ eventually decide v.

Termination: every correct process of $group(p)$ eventually decides exactly one value.

Many different consensus instances may be executed in the same group, leading to several different values being agreed upon. To distinguish the different instances of consensus run within the same group, we assign

to each one of them an unique identifier. Therefore, to propose a value v in a consensus instance of id k, a process p executes *Propose(k,v)*. When such consensus is decided, *Decide(k,v′)* is called back at p, letting p know that the value decided at the instance k was v', which may be equal to v or to something else proposed by some other process of $group(p)$ for instance k.

Although we are considering an asynchronous system, where it is impossible to guarantee the termination of consensus [1], some assumptions can be made regarding the maximum message transmission delay and the existence of a leader in the group. As assuming that the processes know the maximum transmission delay, the considered system model is no longer syncronous, but 'partially synchronous'. In the section 3.5, such assumptions are better detailed.

2.2 Quasi-genuine multicast

Genuine multicast protocols are those where two multicast groups only communicate with each other when one has some message to send to the other. A **quasi-genuine** multicast protocol assumes that:

A1: every group knows from which other groups a multicast message can possibly arrive, and to which other groups it can send a multicast message.

In a quasi-genuine multicast protocol, different groups communicate with each other even if there is no message to be sent from one to the other, but, from *A1*, there is no need for a group g_i to communicate with some other group g_j which cannot send messages to g_i, or receive messages from g_i[1]. This information may be given by the application which is making use of such protocol, so, although not genuine, a quasi-genuine protocol does not imply that each group has to keep sending messages to every other group.

We define $sendersTo(G)$ as the set of other groups which are able to send a message to G. Also, we define $receiversFrom(G)$ as the set of other groups who are able to receive a message from G.

3 Quasi-genuine FIFO multicast

In this section, we present a quasi-genuine multicast protocol which is able to deliver messages in three communication steps when some assumptions hold. Also, an optimistic delivery done within one single communication step is performed; if the optimistic assumption (defined later in this text) holds, such delivery will match the final conservative delivery. If there are mismatches, the application will be responsible for taking action to correct it – i.e. rolling back its state, if it has been changed based on the optimistic delivery.

We want the messages to be delivered according to their sending time from their sources, which is the FIFO order. As every two messages which are delivered in different groups should be delivered in the same order in these groups, we need total order. For the formal definition of the properties of the algorithm, assuming a crash-stop model, we have:

Uniform Validity: if a process multicasts m, then one of the correct processes that is a destination of m eventually delivers m.

Uniform Integrity: for any message m, every correct process p delivers m at most once, and only if some process executed multicast(m) and p is one of m's destinations.

Uniform Agreement: if a process delivers m, then every correct process that is a destination of m also delivers m.

Uniform Total Order: if processes p and p' both deliver m and m', then p delivers m before m' if and only if p' delivers m before m'.

FIFO Order: if a correct process multicasts m before it multicasts m', then no correct process that is a destination of both m and m' delivers m', unless it has previously delivered m.

In section 3.1, we present a minimalistic version of the algorithm, which describes its basic operation, without any optimistic assumptions. The assumptions and optimizations that lead to a conservative delivery within three communication steps, with an optimistic delivery within one single communication step, are presented in sections 3.2 to 3.5. In section 3.6, we discuss the possibility of reducing the number of rollbacks by means of discarding messages – and thus not always ensuring liveness.

[1]This relation may even be asymmetric: it could be possible to send a message from group g_i to g_j, but not in the opposite direction. In this case, in a protocol where a group is blocked waiting for possible messages from other groups, g_j may block its delivery of messages waiting for some kind of "clearance" from g_i, but g_i will never block waiting for messages from g_j.

3.1 Baseline algorithm

Each message m has a source group $m.src$, a set of destination groups $m.dst$ and a timestamp $m.ts$. Note that, by abuse of notation, we have 'process $p \in m.dst$' instead of '$\exists$ group $G \in m.dst$: process $p \in G$'. The total order delivery of messages in a group can be solved by using consensus. Each consensus instance agrees upon some message set as the next ones to be delivered – messages within the same set are told apart by the timestamp $m.ts$ applied by the process which created them; to solve timestamp collisions, the unique id of the sender process can be used. If processes send messages to each other using FIFO reliable channels, and after receival, messages are proposed via consensus in the order in which they are received, the FIFO delivery order is ensured.

The complicating factor is the possibility of a message having at least one destination group different from its source group. So, all involved groups must somehow agree regarding the delivery order of these messages. However, from assumption *A1*, each group knows which other groups it could send messages to – or receive messages from. We can use this by defining *barriers* for multicast, such that $barrier(G_{send}, G_{recv}) = t$ means that the group G_{send} promised that it would send no more messages with a timestamp lower than or equal to t to group G_{recv}. We have defined that $sendersTo(G) = \{G' \neq G : G' \text{ is able to send a message to } G\}$. When a process p, from group G, has received all the barrier values from all the groups in $sendersTo(G)$, and they are all greater than a value t, then p knows that no more messages with timestamp lower than t are coming from other groups and that, once the local ordering (the ordering of messages originated in G) is done, all the pending messages with timestamp up to t can be delivered. Besides, a barrier is sent along with the bundle of all messages with timestamp greater than the last previous barrier sent from G_{send} to G_{recv}, so that when a process has received a barrier from a group, it means that it knows all the messages sent by that group until the time value stored in that barrier.

As mentioned before, we use consensus to deliver messages. Consider that each consensus instance I from each group $I.grp$ receives a monotonically increasing unique integer identifier, without gaps, that is, for any two instances I_i and I_k, such that $I_i.grp = I_k.grp$, if $I_i.id + 1 < I_k.id$, there is necessarily an instance $I_j : I_i.grp = I_j.grp = I_k.grp \wedge I_i.id < I_j.id < I_k.id$. No group runs two consensus instances in parallel: before initiating an instance of id $k+1$ each process checks whether the instance k has already been decided, so some messages may wait to be proposed. When a process is allowed to initiate a new consensus instance, the pending messages may be proposed as a batch.

Since we are using the timestamp given at the creation of a message by its sender, it might occur that, after a message m' has been proposed by a process p in some consensus instance, a message $m : m.ts < m'.ts$ arrives at p. If m is delivered with its original timestamp, the timestamp order is violated and there is no sense in using these timestamps for barriers. There are two possible solutions for that: either the message is simply discarded, and no violation to the timestamp order takes place, or we can change the timestamp of m to something greater than the timestamp of m'. If uniform validity is to be ensured, discarding m is not an option. As we need, then, to change the value of $m.ts$, two things should be noted: first, as we are using reliable FIFO channels, then the process which sent m is different than that which sent m', so inverting their order does not violate FIFO; finally, increasing a message timestamp must be done with caution, so that messages created by different groups at the same time have roughly the same timestamp and the barrier mechanism is efficient – if a long sequence of messages have their timestamps increased, the last one of them may wait a long time until all the barriers required to deliver it have arrived.

To allow for the timestamp of messages to be increased and still have these messages delivered as soon as possible, their timestamps are increased by an infinitesimal value. For that reason, each timestamp value will consist of a real-time clock value, and a sequence value, which is used only when messages need to have their timestamps changed. Therefore, we have that $m.ts = (rtc, seq)$, where rtc is some value related to real-time clock of a process and seq is a sequence number to define an order between messages with the same rtc. Then we have:

$$m.ts < m'.ts \iff m.ts.rtc < m'.ts.rtc \vee (m.ts.rtc = m'.ts.rtc \wedge m.ts.seq < m'.ts.seq)$$

The way each process p of a group G handles each message m will depend on whether G is the source group of m:

- If G is the source of m:

When some process in G multicasts m, it is first fifor-mcast to the other processes of G. Then, when m is fifor-delivered at a process $p \in G$, p checks whether the latest consensus instance I_k in which it participated, or is trying to start, has already been decided – if not, p enqueues m in a *propPending* queue as the next message to be proposed by it in the consensus instance I_{k+1}, so other tasks can keep being executed. Then, once I_k has been decided, p may start a new instance. Before that, all messages in *propPending* that have been already decided are discarded from *propPending*. The rest is proposed as a batch in I_{k+1}.

Once m has been decided, it is not immediately delivered to the application. Instead, p checks whether some message $m' : m'.ts \geq m.ts$ has been decided previously. If that is the case, the value of $m.ts$ is changed to a value greater than the timestamp of any other message previously decided within G. Then, m is inserted into a *barPending* list for later being delivered, which will happen once every group G' in $sendersTo(G)$ has already sent a message $barrier(G', G) = t_b$, such that $t_b \geq m.ts$. Only after that will p know for sure that no message $m'' : m''.ts < m.ts$ will arrive from another group.

After learning that m has been decided, p checks whether there is some other group which is also a destination of m. If that is the case, p fifor-mcasts m to every $p' \in (m.dst \setminus \{G\})$.

- When G is one of the destinations of m, but m was originated in some other group G':

  Although every process in G' sends this message, it is done via a FIFO reliable channel, so even if messages are fifor-delivered twice at p, once m is fifor-delivered, every message $m' : m'.ts < m.ts$ from G has also already been fifor-delivered. As there may be multiple fifor-deliveries of m, only the first one is considered, upon which m is inserted into the *barPending* list of p. Besides, the value of $barrier(G', G)$ is set to $m.ts$, for p knows that no more messages with lower timestamps will arrive from G'.

The messages in the *barPending* list are always sorted in ascending order of their timestamps. When the first message m in the *barPending* list of a process $p \in G$ is such that $m.ts < barrier(G, G')$ for all $G' \in sendersTo(G)$, then m is delivered by p to the application as the next message. We claim that this delivery respects the FIFO total order[2].

A more formal description of the protocol is given in Algorithm 1. We consider that three primitives are given: $getTime()$, which returns the current value of the local wallclock; $Propose(k, val)$, which proposes a value val for the consensus instance of id k within its group; and also $Decide(k, val)$, which is called when the consensus instance of id k finishes. $Decide(k, val)$ is called for all the processes of the group that initiated it, when they learn that the value val has been agreed upon in instance of id k. For the sake of simplicity, we assume that, for consensus instances within the same group, the values are decided in the same order of the instances id's[3]. Finally, we also use a FIFO reliable multicast primitive fifor-mcast(m,*groupSet*), which fifor-delivers m to all the processes in all the groups in *groupSet* in one communication step, in FIFO order (e.g. the one described in [**?**]).

Moreover, each process p of group G keeps some lists of messages:

- *propPending*, containing the messages waiting to be proposed by p;
- *barPending*, with the messages ready to be delivered, but which may be waiting for barriers from the groups in $sendersTo(G)$;
- *decided*, which contains the messages that have already been proposed and decided within $group(p)$;
- *delivered*, which contains the messages that have been delivered already.

Something that must be noticed is that each message m might not have its source group as a destination. Anyway, m still has to be agreed upon in its group of origin G, so that its order among other messages from G may be decided and for m to be retrievable even in the presence of failures.

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 fifor-deliver a message and T_{cons} is the time needed to execute a consensus instance.

[2]Proof needed.

[3]This can be easily done by delaying the callback of $Decide(k, val)$ while there is some unfinished consensus instance of id $k' : k' < k$ from the same group.

Algorithm 1 multicast(m) – executed by every process p from group G

1: Initialization
2: $k \leftarrow 0$, $nextProp \leftarrow 0$, $decided \leftarrow \emptyset$, $delivered \leftarrow \emptyset$, $propPending \leftarrow \emptyset$, $barPending \leftarrow \emptyset$
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: fifor-mcast(m,$\{G\}$)

8: *When* fifor-deliver(m')
9: **if** $G = m'.src$ **then**
10: $propPending \leftarrow propPending \cup \{m'\}$
11: **else if** $m' \notin barPending \wedge m' \notin delivered$ **then**
12: $barPending \leftarrow barPending \cup \{m'\}$
13: $barrier(m'.src, G) \leftarrow m'.ts$

14: *When* $\exists m \in propPending \wedge nextProp = k$
15: $propPending \leftarrow propPending \setminus decided$
16: **if** $propPending \neq \emptyset$ **then**
17: $nextProp \leftarrow k + 1$
18: Propose(k,$propPending$)

19: *When* Decide(k',$msgSet$)
20: **while** $\exists m \in msgSet : (\forall m' \in msgSet : m \neq m' \Rightarrow m.ts < m'.ts)$ **do**
21: $msgSet \leftarrow msgSet \setminus \{m\}$ {*the messages are handled in ascending order of timestamp*}
22: **if** $\exists m' \in decided : m'.ts \geq m.ts \wedge (\nexists m'' \in decided : m''.ts > m'.ts)$ **then**
23: $m.ts \leftarrow (m'.ts.rtc, m'.ts.seq + 1)$
24: $decided \leftarrow decided \cup \{m\}$
25: **if** $G \in m.dst$ **then**
26: $barPending \leftarrow barPending \cup \{m\}$
27: fifor-mcast(m,$m.dst \setminus \{G\}$)
28: $nextProp \leftarrow k' + 1$
29: $k \leftarrow k' + 1$

30: *When* $\exists m \in barPending : \forall G' \in sendersTo(G) : m.ts < barrier(G', G)$
$\wedge \nexists m' \in barPending : m'.ts < m.ts$
31: $barPending \leftarrow barPending \setminus \{m\}$
32: **if** $m \neq null$ **then**
33: deliver(m)
34: $delivered \leftarrow delivered \cup \{m\}$

This value is counted twice because m may have been inserted into *propPending* 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 *propPending* right before some proposal has been made, so it would take only $\delta + T_{cons}$ to decide m. As m may go into *propPending* 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.

3.2 Addressing liveness

The problem with Algorithm 1 is that it does not guarantee liveness when a group has no message to receive from some other group and then keeps waiting for a new message to increase the barrier value and proceed with the delivery of new messages. However, liveness can be easily provided by sending periodic empty messages from G to each $G' \in receiversFrom(G)$ to which no message has been sent for a specified time period. Section 3.2.1 describes such approach. In section 3.2.2, another approach, based on sending empty messages when requested, is presented. The trade-off between them 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.

3.2.1 Sending empty messages periodically

The simplest way to provide liveness to the delivery algorithm is by sending empty messages periodically. The time threshold *barrierThreshold* defines how long a process waits for a meaningful message to be sent before sending an empty message to ensure that some other group is eventually unblocked. Algorithm 2 describes this. When a message m from group G to some other group G' has been fifor-mcast by $p \in G$, p knows that the other processes of G did the same and that m will be eventually received by the processes of G', serving as barrier from G to G' (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 G' with the sole purpose of increasing their barrier values and allow for the delivery of possibly blocked messages in G'.

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

```
1:  Initialization
2:    for all G' ∈ receiversFrom(G) do
3:      lastBarrierCreated(G') = −∞

4:  When fifor-mcasting a message m to some group G' ≠ G
5:    lastBarrierCreated(G') ← m.ts.rtc

6:  When ∃G' ∈ receiversFrom(G) : getTime() − lastBarrierCreated(G') > barrierThreshold
7:    null ← empty message
8:    null.ts ← (lastBarrierCreated(G') + barrierThreshold, 0)
9:    null.src ← G
10:   null.dst ← {G'}
11:   propPending ← propPending ∪ {null}        {saving that nothing was sent until null.ts}
```

The problem with addressing liveness this way is that, in the worst case, G' has decided a message m and has just received the last barrier b from G, such that $b < m.ts$. This would mean that, if G has no messages to send to G', G' will have to wait, at least, for *barrierThreshold* – maybe just to receive some $b' : b' < m.ts$, having to wait again and so on. How long exactly it will take to deliver m depends on many variables, such as how far in the past m was created and how long it takes for some barrier $b : b > m.ts$ to arrive.

It is necessary to guarantee that a *null* message will eventually be proposed, decided, and a barrier will be sent to some group which might be needing it, so that progression is guaranteed. Therefore, this kind of messages are created by every process in the group, since if any one of them does not have it, such message

might never be decided. To prevent the multiple *null* messages – created by different processes within the group – of being decided, they could be created in a way such that the different processes can somehow figure out that two different *null* messages are equivalent[4].

3.2.2 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[5] 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 *propPending* 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.

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 fifor-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 fifor-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 fifor-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.

3.3 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 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.

[4]This could be done by assuming no timestamp collisions and by using them to uniquely identify messages. Then, only one of the messages created with the same timestamp (l. 8 of Algorithm 2) would be decided.

[5]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: $k \leftarrow 0$, $nextProp \leftarrow 0$, $decided \leftarrow \emptyset$, $delivered \leftarrow \emptyset$, $propPending \leftarrow \emptyset$, $barPending \leftarrow \emptyset$
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: fifor-mcast(m,$\{G\}$)

8: *When* fifor-deliver(m')
9: **if** $G = m'.src$ **then**
10: $propPending \leftarrow propPending \cup \{m'\}$
11: **else if** $m' \notin barPending \wedge m' \notin delivered$ **then**
12: $barrier(m'.src, G) \leftarrow m'.ts$
13: **if** $G \in m'.dst$ **then**
14: $barPending \leftarrow barPending \cup \{m'\}$
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: $propPending \leftarrow propPending \cup \{null\}$

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

26: *When* Decide(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** $\exists m' \in decided : m'.ts \geq m.ts \wedge (\nexists m'' \in decided : m''.ts > m'.ts)$ **then**
30: $m.ts \leftarrow (m'.ts.rtc, m'.ts.seq + 1)$
31: **if** $G \in m.dst$ **then**
32: $barPending \leftarrow barPending \cup \{m\}$
33: $decided \leftarrow decided \cup \{m\}$
34: **if** $m \neq null$ **then**
35: fifor-mcast(m,$(m.dst \setminus \{G\}) \cup blockers(m)$) {*the message is sent to blockers(m), as a barrier request*}
36: **else**
37: fifor-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* $\exists m \in barPending : \forall G' \in sendersTo(G) : m.ts < barrier(G', G)$
$\wedge \nexists m' \in barPending : m'.ts < m.ts$
41: $barPending \leftarrow barPending \setminus \{m\}$
42: **if** $m \neq null$ **then**
43: deliver(m)
44: $delivered \leftarrow delivered \cup \{m\}$

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[6]. 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 fifor-delivered at p, instead of proposing it, p inserts it into an *optPending* list, 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.

3.4 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 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

[6]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.

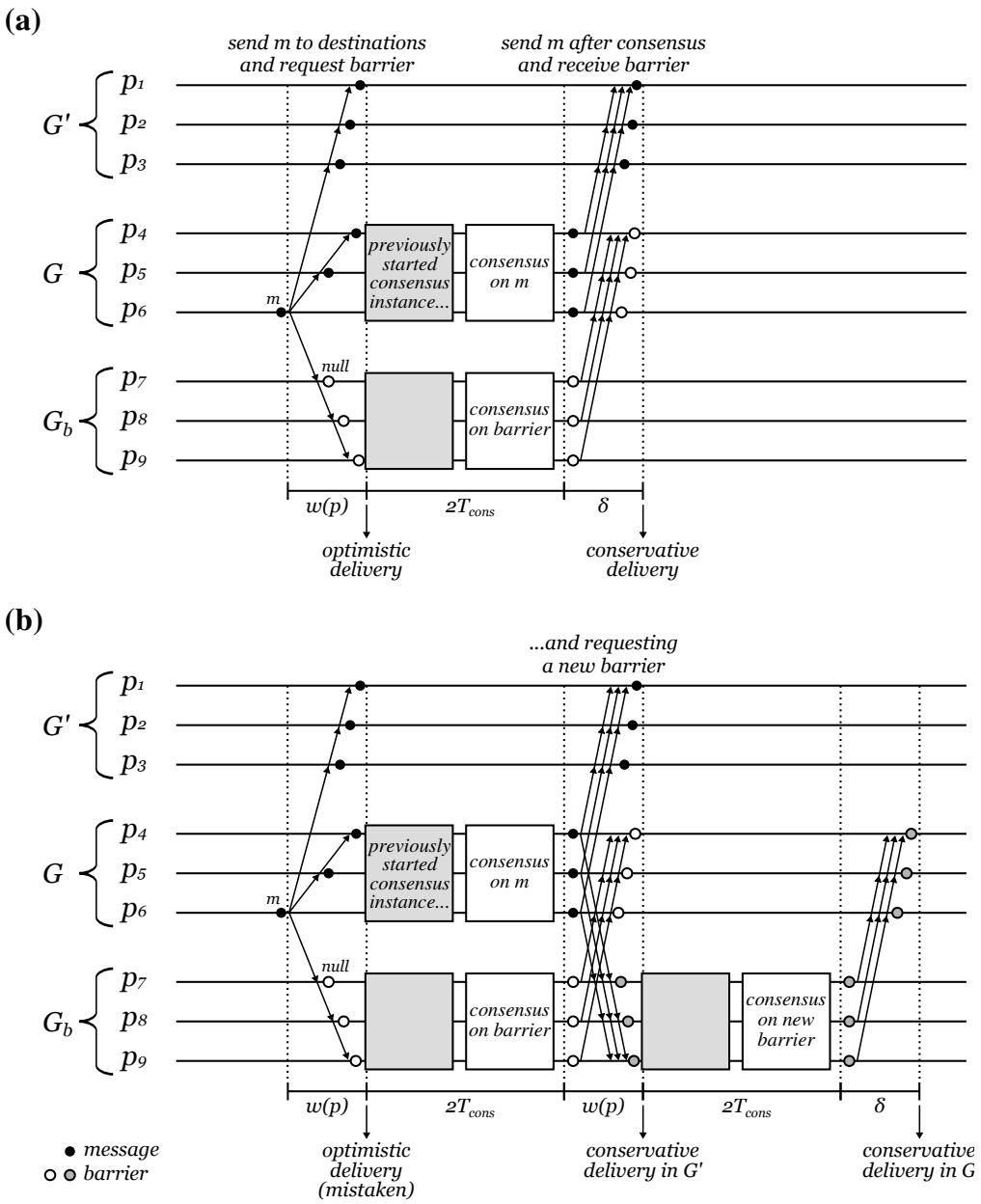

Figure 1: Execution examples when the optimistic assumption holds (a) and when its failure to hold causes a violation of the timestamp order (b)

whether m will be decided in I or not, it keeps the double $(m, I.id)$ in a *trying* list. 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* list. 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()* 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.

3.5 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[7];
- 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 fifor-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 fifor-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 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 fifor-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.

[7] 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.

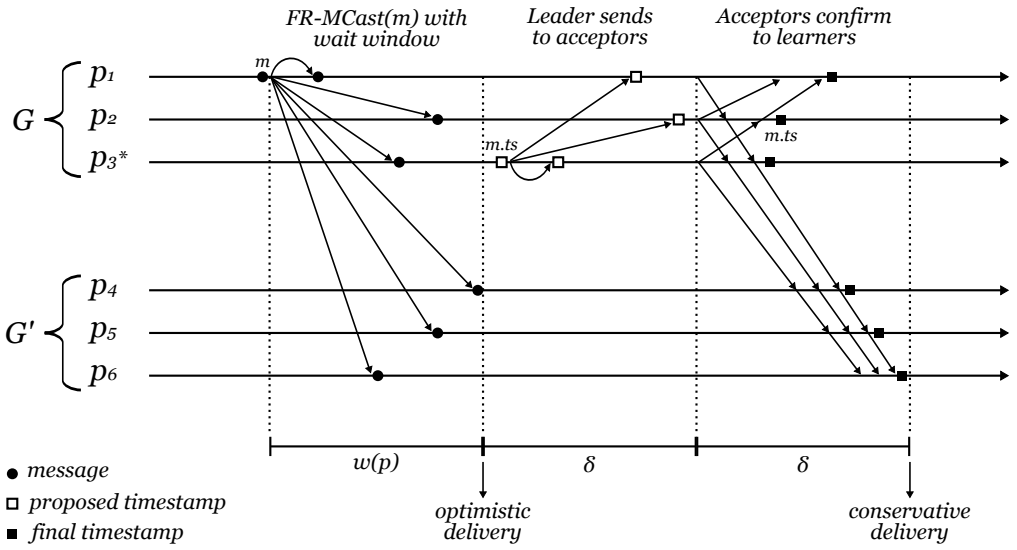

Figure 2: Deciding the final timestamp and notifying other groups in three communication steps

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.

3.6 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.

4 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.

4.1 Uniform Validity

Lemma 1. *Once a message has been multicast, it will be inserted into the barPending list 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 fifor-mcast to all processes in G (l. 7). From the uniform agreement property of the fifor-mcast primitive, every correct process from G fifor-delivers m. Once m is fifor-delivered by each process $p \in G$, it is inserted into the *propPending* list of that process, to be included in the next proposal from p. As every process from G received m and enqueued it for being proposed, and since m is only removed from *propPending* when decided (l. 15), every processes of G will propose it at some point and m will eventually be decided. At this point, if $G \in m.dst$, every correct process of G will insert m into its *barPending* list (l. 26). Besides, each correct process in G fifor-mcasts m to all other destination groups that m might be adressed to.

Let p' be any process in a destination group G' of the message m, such that $G' \neq G$. In this case, once m is fifor-delivered at p', it inserts such message into the *barPending* list, unless it has already done so (lines 11 and 12). □

Lemma 2. *Given a message m in the barPending list of a correct process p, from group G, eventually every group $G' \in sendersTo(G)$ will have sent $barrier(G', G) > m.ts$ to p.*

Proof. Every process p' keeps executing Algorithm 2, thus sending ever-increasing barrier values to all processes in $receiversFrom(group(p'))$. Also, if $G \in receiversFrom(G')$, then $G' \in sendersTo(G)$. This means that, at some point, all correct processes of each group in $sendersTo(G)$ will have inserted a message – be it *null* or not – in their *propPending* lists, with a timestamp greater than, or equal to, that of m. As this is done by every process in each of the groups in $sendersTo(G)$, and a message is removed from *propPending* only when decided, then eventually *null* will be decided and fifor-mcast to G. From the uniform validity property of fifor-mcast, such messages will be delivered by p, which will insert them into its *barPending* list. From the algorithm, the timestamps of the messages are no longer changeable after they have been inserted into *barPending* so, eventually, p will have in such list a barrier with a value greater than or equal to $m.ts$ from each group in $sendersTo(G)$. □

Lemma 3. *If a process p from a group G has received a message m from another group $G' \neq G$, then p has also received any message m' from G', where $m'.ts < m.ts$.*

Proof. Every process p' from G' sends messages to the processes of G, including p, using a FIFO channel, via the fifor-mcast primitive, in the same order in which messages are decided within G'. Also, the agreement property of consensus ensures that all processes from G' will decide the same messages in each consensus instance k. Finally, p' fifor-mcasts to G every message $m : m.src = G' \wedge G \in m.dst$ (l. 27 of Algorithm 1).

Assume, by way of contradiction, that p received a message $m' : m'.ts < m.ts$ from some process $p'' \in G'$ after having received m from p'. As $m'.src = m.src$, this means that p' either will send m' afterwards, contradicting the fact that the processes in G' agree on each batch of messages decided and send them to G in the same FIFO order, or that p' will not send m' at all, contradicting the uniform validity property of the fifor-mcast primitive. Therefore, the assumption is false and the lemma is valid. □

Lemma 4. *Once a message has been inserted into either the barPending list or the delivered list of a given process, it will always belong to one of them.*

Proof. Immediate from the algorithm, considering that a message is never removed from the *delivered* list, and that it is removed from *barPending* (l. 31) right before being inserted into *delivered* (l. 34). □

Lemma 5. *Once a process inserts a message m from some group into its barPending list, no message $m' : m'.ts < m.ts$ from the same group will be inserted into the same barPending list afterwards.*

Proof. Let p be a process from the group G which sent m. Before inserting m into the *decided* list (l. 24 of Algorithm 1), p checks whether there is already some other message with an equal or lower timestamp in such list. If that is the case, the timestamp of m is changed to a value greater than that of any other message already within *decided*. Only then m may be inserted into the $barPending$ list of p (l. 26). Therefore, p inserts messages from other processes in $group(p)$ into its $barPending$ list in ascending order of timestamps. Besides, each process from G fifor-mcasts m to the other destinations of m in this same order (l. 27).

Regarding groups other than the source of the message, let p' be any process that is a destination of m, such that $p' \in G' \wedge G' \neq G$. The process p' inserts m into its $barPending$ list in line 12. Considering that every process in a group sends that group's messages to other groups in the same order in which they are decided, that once a group has received some message from another group, any message prior to that one from the same group has also already been received (from Lemma 3), and that no message is inserted twice in the $barPending$ list of any process (from line 11 and Lemma 4), we can infer that, once p' inserts m into its $barPending$ list, no message $m' : m'.ts < m.ts$ from G will be inserted into the same $barPending$ list afterwards. □

Lemma 6. *Once a message m has been inserted into the $barPending$ list of a process p, it will eventually be removed from such list and, if m is different from null, it will be delivered by p.*

Proof. Eventually, p will have received a barrier from every group in $sendersTo(group(p))$ with a value greater than the timestamp of m (from Lemma 2). Once this happens, as any barrier is also a message, no more messages with a timestamp lower than that of m will be inserted into the $barPending$ list of p (from Lemma 5). Let $barReady$ be the set of m plus all messages from $barPending$ whose timestamps are lower than that of m, that is, $barReady = \{m\} \cup \{m' : m' \in barPending \wedge m'.ts < m.ts\}$.

We can infer that no more messages will be included in this set, meaning that any other message that might be later inserted into $barPending$ will have a timestamp greater than that of any message in $barReady$. Besides, the barrier received for m also apply to all the other messages in this set, since their timestamps are lower than that of m. We can, therefore, prove that all these messages will eventually be delivered by induction on the position i of each message m_i in $barReady$, in ascending order of timestamps.

Base case ($i = 1$): Let m_1 be the first message in $barReady$. It means that $\nexists m \in barPending : m.ts < m_1.ts$. The message m_1 has the lowest timestamp in the list and all the necessary barriers have been received already. Therefore, m_1 satisfies all conditions from line 30 of Algorithm 1 and will be removed from $barPending$. If it is different from *null*, it will also be delivered.

Induction step: Suppose that m_i will eventually be removed from $barPending$. Once this happens, it means that m_i was the first message in the list. Since no more messages have been inserted into $barReady$, as soon as m_i is removed, m_{i+1} will be the new first one. Then, as m_{i+1} will have the lowest timestamp in $barReady$, it will also have the lowest timestamp of $barReady$. As all barriers necessary for m have already been received and $m_{i+1}.ts \leq m.ts$, it will satisfy both conditions of line 30 from Algorithm 1. Thus, m_{i+1} will be removed from $barPending$ – and delivered, if it is different from *null*. □

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 1, Lemma 6, and from the assumption that the application will not multicast a *null* message – not in the multicast protocol level at least; if the application needs to multicast an empty message, in the protocol level it would not be handled as a *null* message. □

4.2 Uniform Integrity

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

Proof. For a message m to be delivered (l. 33), it must be contained in the $barPending$ list (from the condition in line 30). There are two possibilities to when m has been inserted into such list:

- m has been originated in the same same group of p, that is, $p \in m.dst$, which means m was inserted into $barPending$ in line 26 of Algorithm 1, or

- m has been sent from a group $G \neq G'$, which means that it was inserted into *barPending* in line 12.

In line 26, m is inserted into *barPending* only if $G \in m.dst$, which satisfies condition (2), and if it was decided in some consensus instance within G. To have been decided within G, from the properties of consensus [3], we know that it must have been proposed by some process of G, which happens in line 18. In line 18, for a message to be proposed, it must be in the *propPending*, as its contents are the value proposed. For a message to be in the *propPending* list, it must have been inserted there, which happens only in line 10. For l. 10 to be executed, m must have been fifor-delivered and G must be the source group of m, that is, $m.src = G$. For a message to be fifor-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 fifor-mcasts a message to its own group.

Also, for a message to be inserted into *propPending* (l. 10) it cannot have been decided already (l. 15). We assume here that a process can only make a proposal for consensus instance k when it knows all the values decided in all instances prior to k. Therefore, when a process proposes a message, it knows all the messages which were already decided in previous consensus instances. As a decided message is inserted into a *decided* list (l. 24, which is always executed after a decision), a process cannot propose a message which was decided already and, therefore, a message cannot be decided twice (in two different consensus instances). As a message is inserted into *barPending* by processes of its own group of origin only when decided (l. 26), it cannot be inserted twice, thus it cannot be delivered twice, satisfying condition (3).

As for processes from destination groups which are not from the source group of m, they will never execute line 26, since they never decide on m. However, processes of the group $m.src$, which is the source of m, will fifor-mcast m to every G' which is a destination of m other than G (l. 27). When a process in G' fifor-delivers m, it checks whether m was already inserted into *barPending* or *delivered*, where *delivered* is the list of all messages which were ever delivered. Once a message has been inserted into $barPending \cup delivered$, it stays there forever – a message is only removed from *barPending* in line 31, right after what it is then inserted into *delivered*, in line 34. Therefore, also messages which arrive from other groups are never inserted twice into *barPending*. As each message is removed from *barPending* right before being delivered, it is never delivered twice, satisfying (3).

For line 12, m is inserted into *barPending* of process p' of group G' when it has been fifor-delivered, and only if $m.src \neq G'$. For a process $p \in G \neq G'$ to fifor-mcast m, m must have been decided within G, which means that it was proposed within G, therefore it was multicast – satisfying condition (1) – by $G \neq G'$ to G', which, from line 27, 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. □

4.3 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 2, Lemma 1 and Lemma 6. □

4.4 Atomic Order

Lemma 7. *If two correct processes are both destinations of a message m, then they agree on its final timestamp $m.ts$.*

Proof. Let p be the process that is multicasting m. We can prove this lemma by demonstrating that any other process p' will agree with p on the timestamp of m. Let us first consider the case of p' belonging to the same group as p. We can prove by induction on the identifier k of each consensus instance within $group(p)$:

Base case ($k = 1$): As this is the first consensus instance of the group, it means that the $decided = \emptyset$. From the agreement property of consensus, we know that p and p' decided the same contents for $msgSet$ (l. 19). Each message included in such agreed set also includes an initial timestamp field. As the *decided* set is empty, such timestamps are not changed in neither p or p' (l. 22). Therefore, both processes consider the same timestamp value for every message. Besides, as the $msgSet$ is the same for both, the *decided* set remains

identical for both p and p'.

Induction step: Suppose that p and p' have already learnt the decisions of instance k, their *decided* sets remained identical and they decided the same timestamp value of each message sent from some process in their group so far. As we assume that all processes within a group learn the decisions in the same consensus instace identifier order, then both p and p' will next learn the decision of the same 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* set (l. 22). As the $msgSet$ and *decided* sets are each identical in p and p', they will make the exact same change to the timestamp of each message in the *while* loop (l. 20 to l. 27). Therefore, they also agree on the timestamps of each message decided on consensus instance $k+1$ and their *decided* sets remain identical.

Now, consider the case of p' not belonging to the same group of p. Let q be any process of $group(p)$. After setting the final timestamp of m (l. 20 to l. 27), q fifor-mcasts m to $group(p')$ (l. 27). Once p' fifor-delivers m, it never changes the timestamp of m, which already contains the final value, set by q in accordance with p. As any two processes on the same group from which m has been multicast agree on its timestamp, and since any process from other group which is also a destination of m agree on its timestamp as well, than any two processes that are destinations of m agree on its timestamp. □

Lemma 8. *If two messages m and m' have both a correct process p as a destination, and $m.ts < m'.ts$, then p delivers m' after delivering m.*

Proof. Consider Algorithm 1. Suppose, by way of contradiction, that m' has already been delivered by p, whereas m has not. Message m might have already been inserted into $barPending$ or not. In the former case, m' could not have been delivered, since $m.ts < m'.ts$, thus it would not satisfy the condition from line 30 until m has been delivered, so we have a contradiction in the case that m was already in $barPending$.

The other case is if m has not been inserted into $barPending$. From line 30, we know that p has received some barrier b from every group in $sendersTo(group(p))$. Therefore, from Lemma 5, and since a 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 7, Lemma 8 and Proposition 3. □

4.5 FIFO Order

Lemma 9. *If a process p multicasts m, then m', then the final values of their timestamps will be such that $m.ts < m'.ts$.*

Proof. From lines 6 and 7, and the properties of the fifor-mcast primitive, we know that all correct processes fifor-deliver m and then m', which means that each process p' from $group(p)$ also inserts them into its $propPending$ list in this same order.

Suppose, by way of contradiction, that m' is decided before m. This implies that some process p proposed a message set that included m', but not m. As p inserted m before m' in its $propPending$ list, it means that m was removed by p from such list before being included in some proposed message set. However, the only possibility of removing a message from $propPending$ is on line 15, which means that m already belonged to the *decided* set, so it had been decided before m', which is a contradiction. Therefore, m' is not decided before m.

This also means that m' is not inserted into the *decided* list before m has already been inserted. From line 6, we know that the initial timestamp of m is already smaller than that of m'. The only way to change this order is by making the final timestamp $m.ts$ bigger than $m'.ts$ on line 23. Even if this happens, as m' is inserted into *decided* after m, from line 22, we know that the final timestamp of m' will be made greater than that of m. □

Proposition 5. *If p multicasts m and then m', then no process p' in both $m.dst$ and $m'.dst$ delivers m' before delivering m.*

Proof. Immediate from Lemma 8 and Lemma 9. □

5 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 $p \dots$). sequencer based (no tolerance for failures of the sequencer). not mentioning multicast.

6 Experimental results

7 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.