A Quasi-Genuine FIFO Total Order Multicast Primitive

June 10, 2011

Abstract

1 Introduction

Some multicast primitives have been devised in a 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 a given group. 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 nodes distributed over a geographical area. Nodes may fail by crashing and subsequently recover, but do not experience arbitrary behavior (i.e., no Byzantine failures). Communication is done by message passing, through the primitives $\mathit{send}(p, m)$ and $\mathit{receive}(m)$, where p is the addressee of message m. Messages can be lost but not corrupted. If a message is repeatedly resubmitted to a *correct node* (defined below), it is eventually received.

Our protocols ensure safety under both asynchronous and synchronous execution periods. The FLP impossibility result [2] states that under asynchronous assumptions consensus cannot be both safe and live. We thus assume that the system is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the *Global Stabilization Time (GST)* [1], and it is unknown to the nodes.

Before GST, there are no bounds on the time it takes for messages to be transmitted and actions to be executed. After GST, such bounds exist but are unknown. After GST nodes are either *correct* or *faulty.* A correct node is operational "forever" and can reliably exchange messages with other correct nodes. This assumption is only needed to prove liveness properties about the system. In practice, "forever" means long enough for one instance of consensus to terminate.

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 primitive, so, although not genuine, a quasi-genuine protocol does not imply that each group has to keep sending messages to every other group.

[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.

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.

As messages are delivered according to their generation time, the delivery order is FIFO. 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 QGFTO-Mcasts m, then one of the correct processes that is a destination of m eventually QGFTO-Delivers m.

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

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

Uniform Total Order: if processes p and p' both QGFTO-Deliver m and m', then p QGFTO-Delivers m before m' if and only if p' QGFTO-Delivers m before m'.

FIFO Order: if a correct process QGFTO-Mcasts m before it QGFTO-Mcasts m', then no correct process that is a destination of both m and m' QGFTO-Delivers m', unless it has previously QGFTO-Delivered m.

Here, we consider that there are several processes. Each process p belongs to a group $G = group(p)$, that is, $p \in G = group(p)$.

3 Message delivery 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 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 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.

However, this implies using the timestamp given at the creation of a message by it sender. Therefore, it might be the case that, after a message m' has been proposed by a process p in a 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'. We want to ensure uniform validity, so 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 the wallclock (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:

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

There are three possibilities for each message m, in the perspective of a process p of G:

- The message m was originated in G, which is the only destination of m:

  In this case, when m is received by p, 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 being 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 is decided, it is not immediately delivered to the application. Instead, p checks whether some message $m_{prv} : m_{prv}.ts \geq m.ts$ 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 $barrier(G', G) = t$, such that $t > m.ts$. This is done because there could be a message m' yet to come from another group G', such that $m'.ts < m.ts$.

- When m is originated in G, but it has at least one group other than G as a destination:

  In this case, when m is received, p tries to initiate a consensus instance within G to decide m. If p cannot start the proposal now, m is enqueued in *propPending* for being proposed later along with other pending messages. Then, once p may start a new instance, all messages in *propPending* that have been already decided are discarded from *propPending*. The rest is proposed as a batch in the new consensus instance. Once any message m is decided, p checks whether some message $m_{prv} : m_{prv}.ts > m.ts$ 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, if $G \in m.dst$, m is inserted in the *barPending* list. Besides, when m is decided, p sends m to every $p' \in (m.dst \setminus \{G\})$. When m is received by each $p' \in G'$, p' checks whether it has ever inserted m in its own *barPending* list. If not, p' inserts m into *barPending* and adjusts $barrier(G, G')$ to $m.ts$. To ensure that, once a message m is received from another group G, every message $m' : m'.ts < m.ts$ also from G has already been received, every message is sent through a lossless FIFO channel[2].

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

  In this case, when m is received for the first time[3] by p, m is inserted into the *barPending* list of p and the value of $barrier(G', G)$ is set to $m.ts$.

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

[2]An ordinary TCP connection would be enough to provide such FIFO lossless channel. Here, we use FIFO reliable multicast.
[3]Multiple processes may have sent m. To ensure integrity and order, only the first delivery is considered.

$G' \in \mathit{sendersTo}(G)$, then m is QGFTO-Delivered by p to the application as the next message. We claim that this delivery respects the FIFO total order[4].

A more formal description of the protocol is given in Algorithm 1. We consider that three primitives are given: $\mathit{getTime}()$, which returns the current value of the local wallclock; $\mathit{Propose}(k, \mathit{val})$, which proposes a value *val* for the consensus instance of id k within its group; and also $\mathit{Decide}(k, \mathit{val})$, which is called when the consensus instance of id k finishes. $\mathit{Decide}(k, \mathit{val})$ 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[5]. Finally, we also use a FIFO reliable multicast primitive FR-MCast(m,*groupSet*), which FR-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 QGFTO-Delivered, but which may be waiting for barriers from the groups in $\mathit{sendersTo}(G)$;
- *decided*, which contains the messages that have already been proposed and decided within $\mathit{group}(p)$;
- *delivered*, which contains the messages that have been QGFTO-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 QGFTO-Mcast 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 *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 QGFTO-Deliver m will depend on when barriers are received from other groups.

3.1 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 $G' \in \mathit{receiversFrom}(G)$ to which no message has been sent for a specified time period. Section 3.1.1 describes such approach. In section 3.1.2, 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.

3.1.1 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 FR-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

[4]Proof needed.

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

Algorithm 1 QGFTO-Mcast(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 QGFTO-Mcast 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:     $propPending \leftarrow propPending \cup \{m'\}$
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:     if $G \in m.dst$ then
25:       $barPending \leftarrow barPending \cup \{m\}$
26:     $decided \leftarrow decided \cup \{m\}$
27:     FR-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:   QGFTO-Deliver($m$)
33:   $delivered \leftarrow delivered \cup \{m\}$
```

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 $G' \in receiversFrom(G)$ do
3:      $lastBarrierCreated(G') = -\infty$

4:  When FR-MCasting a message $m$ to some group $G' \neq G$
5:    $lastBarrierCreated(G') \leftarrow m.ts.rtc$

6:  When $\exists G' \in receiversFrom(G) : getTime() - lastBarrierCreated(G') > barrierThreshold$
7:    $null \leftarrow$ empty message
8:    $null.ts \leftarrow (lastBarrierCreated(G') + barrierThreshold, 0)$
9:    $null.src \leftarrow G$
10:   $null.dst \leftarrow \{G'\}$
11:   $propPending \leftarrow propPending \cup \{null\}$        {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 QGFTO-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[6].

3.1.2 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[7] 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,

[6]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.

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

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 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.

3.2 Optimistic delivery

We can further decrease the delivery time of the messages. If the timestamp assigned to each message by its sender process does not change, it can be used to predict the final delivery order when such message is FR-Delivered by some of its destinations. However, messages from different destinations might arrive out of order at a destination p_d. However, since a reliable multicast primitive is being used, if every message is first FR-MCast to every destination, and if p_d waits long enough to deliver some message m, every message $m' : m'.ts < m.ts$ will arrive eventually and p_d will be able to deliver them in 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.

We need to mention the clock deviation in this assumption because, even if a process waits for a period longer than the time needed for a message m to arrive since it was sent, it might be the case that a clock deviation between the processes caused the timestamp of m to be too low and lead to a violation of the timestamp order.

There would be now two deliveries for each message: a conservative one, which may follow one of the algorithms described earlier, and an optimistic one. The difference is that, after a message m was FR-Delivered at p, p inserts it into an *optPending* list, which is sorted in ascending order of the timestamps of the messages in it. Before OPT-Delivering (optimistically delivering) m, p waits until its own wallclock has a value greater than $m.ts.rtc + w(p)$. If *A2* holds, then any message from the sender of m received after that will have a timestamp greater than $m.ts$, so m can be delivered.

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 $blockers(m)$. All can be done in one single communication step: sending the message to other processes in $m.src$ to then start a consensus instance, sending it to be optimistically delivered to processes in destination groups other than $m.src$ and sending it to groups in $blockers(m)$.

If it does not hold all the time and some message m is received by a process p after a message $m' : m'.ts > m.ts$ has been already OPT-Delivered 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 a 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,

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

```
Initialization
    $k \leftarrow 0$, $nextProp \leftarrow 0$, $decided \leftarrow \emptyset$, $delivered \leftarrow \emptyset$, $propPending \leftarrow \emptyset$, $barPending \leftarrow \emptyset$
    for all $G' \in sendersTo(G)$ do
        $barrier(G', G) \leftarrow -\infty$

To QGFTO-Mcast a message $m$
    $m.ts \leftarrow (getTime(), 0)$
    FR-MCast($m$,$\{G\}$)

When FR-Deliver($m'$)
    if $G = m'.src$ then
        $propPending \leftarrow propPending \cup \{m'\}$
    else if $m' \notin barPending \wedge m' \notin delivered$ then
        $barrier(m'.src, G) \leftarrow m'.ts$
        if $G \in m'.dst$ then
            $barPending \leftarrow barPending \cup \{m'\}$
        if $G \in blockers(m') \wedge m' \neq null$ then
            $null \leftarrow$ empty message
            $null.src \leftarrow G$
            $null.ts \leftarrow m'.ts$
            $null.dst \leftarrow m'.dst \cap receiversFrom(G)$
            $propPending \leftarrow propPending \cup \{null\}$

When $\exists m \in propPending \wedge nextProp = k$
    $propPending \leftarrow propPending \setminus decided$
    if $propPending \neq \emptyset$ then
        $nextProp \leftarrow k + 1$
        Propose($k$,$propPending$)

When Decide($k$,$msgSet$)
    while $\exists m \in msgSet : (\forall m' \in msgSet : m \neq m' \Rightarrow m.ts < m'.ts)$ do
        $msgSet \leftarrow msgSet \setminus \{m\}$      {the messages are handled in ascending order of timestamp}
        if $\exists m' \in decided : m'.ts \geq m.ts \wedge (\nexists m'' \in decided : m''.ts > m'.ts)$ then
            $m.ts \leftarrow (m'.ts.rtc, m'.ts.seq + 1)$
        if $G \in m.dst$ then
            $barPending \leftarrow barPending \cup \{m\}$
        $decided \leftarrow decided \cup \{m\}$
        if $m \neq null$ then
            FR-MCast($m$,$(m.dst \setminus \{G\}) \cup blockers(m)$)      {the message is sent to blockers(m), as a barrier request}
        else
            FR-MCast($m$,$m.dst$)      {no need to unblock null, besides it would create infinite messages}
    $nextProp \leftarrow k + 1$
    $k \leftarrow k + 1$

When $\exists m \in barPending : \forall G' \in sendersTo(G) : m.ts < barrier(G', G)$
$\wedge \nexists m' \in barPending : m'.ts < m.ts$
    $barPending \leftarrow barPending \setminus \{m\}$
    if $m \neq null$ then
        QGFTO-Deliver($m$)
    $delivered \leftarrow delivered \cup \{m\}$
```

as a barrier request was done in parallel with the conservative delivery algorithm, it will take $2w(p) + 2T_{cons}$ to QGFTO-Deliver m in the worst case. If it does not hold, after $w(p)$, m may be OPT-Delivered in an invalid order and, once 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 $2w(p) + 4T_{cons} + \delta$.

3.3 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 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* 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.

3.4 Using Paxos with a leader for consensus

Although ensuring termination of consensus in an asynchronous system is not possible [2], the Paxos [3] algorithm can guarantee the termination of an instance I as long as some assumptions are held:

- when I starts, a leader has been – or is being – elected and it does not fail for some time[8];
- 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, the leader process 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 QGFTO-Mcast 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 included in the conservative message delivery time we analysed before: instead of waiting for a message to be decided and only then notify other groups, the acceptors of a group can also send the confirmation to processes in other groups, so that also they can

[8]Lamport demonstrates in [3] that, if the time needed to elect a leader is T_{el} and, for a communication step, it is δ, the time needed to conclude a consensus instance would be at most $T_{el} + 9\delta$.

infer the first group's decision right away. This implies that, for each consensus instance run within a group G, each the processes in the groups of $receiversFrom(G)$ would be a learner for that instance. Figure **??** illustrates this: the group G is delivering a message m and the last phase of the consensus protocol already notifies the processes in $G' \in receiversFrom(G)$.

4 Proof of correctness

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...). sequencer based (no tolerance for failures of the sequencer). not mentioning multicast.

6 Experimental results

Really necessary?

7 Conclusion

References

[1] Dwork, C., Lynch, N., and Stockmeyer, L. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM) 35*, 2 (1988), 288–323.

[2] 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.

[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.