A Quasi-Genuine FIFO Total Order Multicast Primitive

June 7, 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 $send(p, m)$ and $receive(m)$, 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.

3 Quasi-genuine Global FIFO 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

[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.

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.

We define $sendersTo(G)$ as the set of groups which are able to send a message to G. Also, we define $receiversFrom(G)$ as the set of 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, $p \in G = group(p)$.

3.1 Message delivery

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 $sendersTo(G) = \{G' : G'$ 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: 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 cause a group to wait a long time until it has all the barriers required to deliver it.

To allow for the timestamp of messages to be increased and still have them 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:

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

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 $m_{prv} : m_{prv}.ts > m.ts$ 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 $p : group(p) = G$, is such that $m.ts < barrier(G, G')$ for all $G' \in sendersTo(G)$, then m is conservatively 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: $getTime()$, 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

[2] An ordinary TCP connection would be enough to provide such FIFO lossless channel.

[3] Multiple processes may have sent m. To ensure integrity and order, only the first delivery is considered.

[4] Proof needed.

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 $sendersTo(G)$;
- *decided*, which contains the messages that have already been proposed and decided within $group(p)$;
- *delivered*, which contains the messages that have been QGFTO-Delivered already.

Something that must be noticed is that some messages might not have their source groups as a destination. Anyway, each message m of this type 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 m will 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.

3.1.1 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 $G' \in receiversFrom(G)$ to which no message has been sent for a specified time threshold *barrierThreshold*. 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. 13 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'.

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. Besides, they are excluded from the *propPending* list only when decided in the group (l. 15 of Algorithm 1). Since they are obviously never delivered to the application, they can remain in such list and be decided even after some message with higher timestamp is decided. Not removing them unless they are decided ensures that they will be sent and that other groups will be able to increase their barrier values. 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].

However, there is a way to provide liveness withouth such a possibly long delay for the conservative delivery of messages, although that would imply creating more messages and making deeper changes in the delivery algorithm.

[5]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.

[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. 9 of Algorithm 2) would be decided.

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 send a message m* – QGFTO-Mcast(m)
6: $m.ts \leftarrow getTime()$ {*current wallclock value as the timestamp of m*}
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: $decided \leftarrow decided \cup msgSet$
21: **for all** $m \in msgSet : G \in m.dst$ **do**
22: $barPending \leftarrow barPending \cup \{m\}$
23: **while** $\exists m \in msgSet : (\forall m' \in msgSet : m \neq m' \Rightarrow m.ts < m'.ts)$ **do**
24: FR-MCast(m,$m.dst \setminus \{G\}$)
25: $msgSet \leftarrow msgSet \setminus \{m\}$ {*the messages are sent in ascending order of timestamp*}
26: $nextProp \leftarrow k + 1$
27: $k \leftarrow k + 1$

28: *When* $\exists m \in barPending : \forall G' \in sendersTo(G) : m.ts < barrier(G', G)$
$\wedge \nexists m' \in barPending : m'.ts < m.ts$
29: $barPending \leftarrow barPending \setminus \{m\}$
30: QGFTO-Deliver(m)
31: $delivered \leftarrow delivered \cup \{m\}$

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: **for all** $G' \in m.dst : G' \neq G$ **do**
6: $lastBarrierCreated(G') \leftarrow m.ts$

7: *When* $\exists G' \in receiversFrom(G) : getTime() - lastBarrierCreated(G') > barrierThreshold$
8: $null \leftarrow$ empty message
9: $null.ts \leftarrow lastBarrierCreated(G') + barrierThreshold$
10: $null.src \leftarrow G$
11: $null.dst \leftarrow \{G'\}$
12: $propPending \leftarrow propPending \cup \{null\}$ {*saving that nothing was sent until null.ts*}

Let $blockers(m)$ be defined as the set of groups whose barrier is needed in order for some group to deliver m. More formally, $blockers(m) = \{G_B : \exists G_{dst} \in m.dst \wedge G_B \in sendersTo(G_{dst})\}$. The idea is that, once each group G_B in $blockers(m)$ have sent 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 has a message m to send, p sends m to every $p' \in m.dst \cup G \cup blockers(m)$. It is sent to the groups in $blockers(m)$, so that they know that there is a message which will be blocked until they send a proper barrier to unblock it.

When the process p' of some group G' receives m, such that $m.src \neq G'$, p' knows that there might be other groups depeding 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 *optPending* queue – to ensure that, once it goes to *propPending*, any message $m' : m'.ts < null.ts \wedge m'.src = G'$ is already there. As soon as $null.ts > now - w(p')$, the *null* message will be proposed in G' and, once *null* is decided, each process of G' will send a $\{null, \text{'cons'}\}$ message to each 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.

Now, assuming that $w(p)$ is the same for every process p, we have a worst case delivery latency for m of $w(p) + 2T_{cons}$, as the *null* message is created and proposed in parallel with m. However, each sender process is sending m not only to every destination process, but also to every process in its group and to every process belonging to some group of $blockers(m)$. This can create a fairly high amount of messages. 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. Although this kind of messages are excluded from the *propPending* list only when decided in the group (l. 26 of Algorithm 3), if at least one of the processes does not have it, it could never be decided. Again, like in Algorithm 2, there might be many processes in a group proposing a *null* message because of the same m. Although this is not wrong, it is inefficient, and a way to identify *null* messages created for the same purpose should be devised[7].

3.2 Recovering from mistakes

Unfortunately, even with a very good delay estimation (e.g. on an environment with a low jitter), there is absolutely no guarantee that the multicast protocol described in section 3 will deliver the game command messages optimistically and conservatively in the same order. When it doesn't, it is considered a *mistake*. Every mistake of the optimistic delivery – either a lost command message, or an out-of-order delivery – will cause a rollback of the optimistic state of the objects and re-execution of some of the optimistically delivered commands.

To perform that, we consider that each object has an optimistic delivery queue, Q_{opt}. Whenever a command is optimistically delivered, the optimistic state is updated and the command is pushed in the back of Q_{opt}. Whenever a command C_c is conservatively delivered, it updates the conservative state of each object in $obj(C_c)$ and, for each

[7]Again, we could use timestamps as unique identifiers and assume no timestamp collisions. But for that to work, instead of making $null.ts = m.ts$ (l. 12 of Algorithm 3), we could define $null.ts$ as a little bit greater than $m.ts$.

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

1: Initialization
2: $k \leftarrow 0$, $nextProp \leftarrow 0$, $decided \leftarrow \emptyset$, $propPending \leftarrow \emptyset$, $optPending \leftarrow \emptyset$, $barPending \leftarrow \emptyset$
3: **for all** $G' \in sendersTo(G)$ **do**
4: $barrier(G', G) \leftarrow -\infty$

5: *To send a message m* – QGFTO-Mcast(m)
6: $m.ts \leftarrow getTime()$ {*current wallclock value as the timestamp of m*}
7: FR-MCast(m,$m.dst \cup \{G\} \cup blockers(m)$)

8: *When* FR-Deliver(m')
9: **if** $G \neq m'.src \wedge \exists G' \in (m'.dst \cap receiversFrom(G))$ **then**
10: $null \leftarrow$ empty message
11: $null.src \leftarrow G$ {*some other group needs a barrier from this one*}
12: $null.ts \leftarrow m'.ts$
13: $null.dst \leftarrow m'.dst \cap receiversFrom(G)$
14: $optPending \leftarrow optPending \cup \{null\}$
15: **if** $m'.ts < getTime() - w(p) \vee G \notin m'.dst$ **then**
16: discard m' {*late commands probably lead to out-of-order delivery*}
17: **else**
18: $optPending \leftarrow optPending \cup \{m'\}$

19: *When* $\exists m \in optPending : getTime() > m.ts + w(p) \wedge \nexists m' \in optPending : m'.ts < m.ts$
20: $optPending \leftarrow optPending \setminus \{m\}$
21: **if** $G \in m.dst \wedge m \neq null$ **then**
22: GF-OPT-Deliver(m)
23: **if** $G = m.src$ **then**
24: $propPending \leftarrow propPending \cup \{m\}$

25: *When* $\exists m \in propPending \wedge nextProp = k$
26: $propPending \leftarrow propPending \setminus (decided \cup \{m' : \exists m'' \in decided \wedge m'.ts < m''.ts \wedge m' \neq null\})$
27: **if** $propPending \neq \emptyset$ **then**
28: $nextProp \leftarrow k + 1$
29: Propose(k,$propPending$)

30: *When* Decide(k,$msgSet$)
31: $decided \leftarrow decided \cup msgSet$
32: **for all** $m \in msgSet : G \in m.dst \wedge m \neq null$ **do**
33: $barPending \leftarrow barPending \cup \{m\}$
34: **while** $\exists m \in msgSet : (\forall m' \in msgSet : m \neq m' \Rightarrow m.ts < m'.ts)$ **do**
35: **for all** $p' \in m.dst \setminus \{G\}$ **do**
36: send(p', $\{m, \text{`cons'}\}$) {*this message is sent through a FIFO lossless channel*}
37: $msgSet \leftarrow msgSet \setminus \{m\}$ {*the messages are sent in ascending order of timestamp*}
38: $nextProp \leftarrow k + 1$
39: $k \leftarrow k + 1$

40: *When* receive($\{m', \{\text{`cons'}\}\}$)
41: **if** $m' \notin barPending \wedge m' \notin delivered \wedge m' \neq null$ **then**
42: $barPending \leftarrow barPending \cup \{m\}$
43: $barrier(m'.src, G) \leftarrow max(m'.ts, barrier(m'.src, G))$ {*channels are FIFO, but there are different senders*}

44: *When* $\exists m \in barPending : \forall G' \in sendersTo(G) : m.ts < barrier(G', G)$
$\wedge \nexists m' \in barPending : m'.ts < m.ts$
45: $barPending \leftarrow barPending \setminus \{m\}$
46: QGFTO-Deliver(m)
47: $delivered \leftarrow delivered \cup \{m\}$

one of them, the algorithm checks whether it is the first command in Q_{opt}. If it is, C_c is simply removed from Q_{opt} and the execution continues. If it isn't, it means that C_c was either optimistically delivered out of order, or it was simply never optmistically delivered. It then checks whether Q_{opt} contains C_c. If it does, it means the command was optimistically deliverd out of order, and it is removed from the list – if Q_{opt} doesn't contain C_c, it was probably lost[8]. Then, the optimistic state is overwritten with the conservative one and, from that state, all the remaining comands in Q_{opt} are re-executed, leading to a new optimistic state for that object.

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.

[8] Also, when C_c is delivered, but it is not in Q_{opt}, the remaining possibility is the very unlikely case where the conservative delivery happened before the optimistic one. To handle this case, C_c is stored in a list of possibly delayed optimistic delivery and, if it is ever optimistically delivered, the algorithm will know that it should only discard that command, instead of updating the optimistic state.