Practical consistency management for geographically distributed MMOG servers

May 27, 2011

Abstract

1 Introduction

2 System model and definitions

We assume a system composed of nodes, divided into *players* and *servers*, 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 [**?**] 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)* [**?**], 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.

A game is composed of a set of objects. The game state is defined by the individual states of each one of its objects. We assume that the game objects are partitioned among different servers. Since objects have a location in the game, one way to perform this partitioning is by zoning the virtual world of the game. Each partition consists of a set of objects of the game and the server responsible for them is their *coordinator*. As partitions represent physical regions in the

game world, we define *neighbor* regions as those which share a border with each other. We consider that each server is replicated, thus forming several groups which consist of the coordinator and its replicas. Therefore, for each region of the game world, there is a group assigned to it. This way, a group is said to coordinate an object when the group's coordinator is the coordinator for that object. Finally, groups are called neighbors when they are assigned to neighbor regions. From now on, the word 'server' will be used for any kind of server, be it a coordinator or a replica.

Each player may send his command to one of the servers – which might not be the group's coordinator, if that provides a lower delay between the issuing of a command and its delivery. In the case of avatar based virtual environments, and as an avatar is usually also an object, the server to which a player is connected belongs to the group of his avatar's coordinator. A command $C = \{c_1, c_2, ...\}$ is composed of one or more subcommands, one subcommand per object it affects. We refer to the set of objects affected by command C as $obj(C)$. Also, we refer to the objects coordinated by a server S as $obj(S)$. Finally, we define $obj(C, S) = \{o : o \in obj(C) \text{ and } o \in obj(S)\}$.

Our consistency criterion is "eventual linearizability". (I'm not sure this is indeed what we want and how to define it, but we do need some consistency criterion...)

3 Protocol

To ensure reliability despite server crashes, each server is replicated using state-machine replication, implemented with Paxos. Each player sends his commands to the server to which he is connected, which then proposes that command in a consensus instance. Each command is assigned a timestamp and executed against objects in timestamp order. We implement this by using a logical clock in each server group. Guaranteeing that the same set of commands is executed by the respective affected objects in the same order provides the level of consistency we are seeking. Therefore, the challenge is how to assign timestamps to commands such that consistency is not violated and commands are not discarded due to stale timestamp values.

However, providing such level of consistency may prove to be costly in an MMOG context, since there may be several communication steps between the sending of a command and its atomic delivery. For this reason, we use a primitive we call quasi-genuine global fifo total order multicast, which is described in section 3.1.

3.1 Quasi-genuine Global FIFO Total Order Multicast

To reduce the time needed to deliver a message, we use a multicast primitive which delivers messages optmistically, based on the time when they are created in a sender. This *optimistic* delivery, although not guaranteeing that all replicas

will receive all messages, is designed to have a fairly low latency, counting from when each message is sent until it is delivered.

The final – conservative, fault tolerant, but costly in terms of communication steps for each message – delivery order should be as close as possible to the optimistic one, so that no rollbacks would be deemed necessary. To explain how this works, we first should understand the idea behind the optimistic delivery. Also, we must define what "quasi-genuine" and "global FIFO" means.

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.

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. Besides, as every two messages which are delivered in different groups should be delivered in the same order in these groups, we need total order. However, FIFO total order is not enough to guarantee one property that we need: let the send time $st(m)$ of a message m be the value of the wallclock of p when m was sent, where p is the process which sent it. We need that, if $st(m_1) < st(m_2)$, then m_1 should be delivered before m_2, if m_1 and m_2 are delivered by the same process. Although this property may be hard to observe, it is used by the optimistic delivery protocol, so that messages can be delivered in order in one communication step, in the best case. If, when each process sends a message m, it stamps the message with the current value $st(m)$ of its wallclock, we define:

Global FIFO order: if processes p_i and p_j send respectively m_i and m_j to the same destination p_d, then if $st(m_i) < st(m_j)$, m_i is not delivered after m_j in p_d.

Here, we consider that there are several processes. Each process p belongs to a group $G = group(p)$, that is, $p \in G = group(p)$. Each group G has a special leader process c, which is its coordinator.

[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.

3.1.1 Optimistic delivery

The basic idea of the optimistic ordering is the following: assuming that the processes have a synchronized clock[2], whenever a process p (either coordinator or not) receives a message m from a client, it immediately applies a timestamp ts to it, which consists simply of the current value of p's wallclock, now. Therefore, $m.ts = now$. A wait window of length $w(p)$ is considered, where $w(p)$ is defined as the highest value of the estimated communication delay plus the wallclock deviation between the process p and any of the other processes in its group G or in any of the groups in $sendersTo(G)$.

More formally, let $\delta(p', p)$ be the maximum time for a message from p' to arrive at p. Also, let $\epsilon(p', p)$ be the deviation between the wallclocks of p and p'. The value of $w(p)$ is estimated as the maximum value of $(\delta(p', p) + \epsilon(p', p))$ for every p' in $G = group(p)$ or in some group of $sendersTo(G)$.

After applying the timestamp to m, the process p immediately forwards it to all the other processes involved, including those in other groups. A process is involved with a message m when it is one of its destinations, which can be inferred from $m.dst$. Then, p puts m in an *optPending* list, where it stays until $now > m.ts + w(p)$, which means that m has been in that list for a time longer than the defined wait window. In the meantime, other messages, sent from other processes, may have been received and also inserted in that list, always sorting by their timestamps.

If $w(p)$ has been correctly estimated, and no message was lost, than all the messages that were supposed to be delivered before m have necessarily been received already. If the same has occurred for all the processes, then all of them have received all the messages, and can deliver them in the same order of ts.

To avoid out-of-order deliveries, when a message m arrives too late, that is, if when m arrives at p, $m.ts < now - w(p)$, the message is simply discarded by p, since another message $m' : m'.ts > m.ts$ may have already been delivered[3].

However, even if the optimistic order is the same for all the processes, it won't be valid if the conservative order is different from it. For that reason, we devised a way to make the conservative delivery order as close as possible to the optimistic one.

3.1.2 Conservative delivery

Instead of having the conservative delivery done completely independently from the optimistic one, we can actually use the latter as a hint for the final delivery order. Since the optimistic delivery should be fast when compared to the conservative one, waiting for it should not decrease the system performance significantly. Also, if we wait a short period longer, we can avoid a rollback later

[2]We don't require here perfectly synchronized clocks, as the optimistic protocol tolerates mistakes by its very definition. We only need clocks which are synchronized enough, so that our delivery order prediction succeeds and matches the conservative delivery order.

[3]We could make this in a way such that m is only discarded by p if, in fact, there was a delivered message $m' : m'.ts > m.ts$. If there was no such message, m could still be delivered without violating the order we defined.

caused by mismatches between the two delivery orders, which is not desirable. The basic idea is, then, to pick the optimistic delivery order seen at the processes of each group.

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 $sendersTo(G) = \{G' : G' \text{ is able to send a message to } G\}$. 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 conservatively 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.

We use consensus to conservatively deliver each message. 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 $I_j : I_i.grp = I_j.grp = I_k.grp \wedge I_i.id < I_j.id < I_k.id$. Consider also that each message sent by a group G has a group sequence number $m.gs$ related to the order in which it is conservatively delivered, relatively to other messages also sent by G. As messages are conservatively delivered via consensus, the group sequence number of a message is equal to the id of the instance in which it was decided, that is, $m.gs = i \Leftrightarrow \exists I : I.grp = m.src \wedge Decide(I.id, m) \wedge I.id = i$.

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 optimistically delivered by p, p checks whether the latest consensus instance I_{prev} in which it participated, or is trying to start, has already been decided – if not, p enqueues m in a queue *propPending* as the next message being proposed by it, so other tasks can keep being executed. Then, once I_{prev} has been decided, p checks wheter m – or a message $m' : m'.ts > m.ts$ – has already been conservatively delivered in some previous consensus instance. If such message was indeed delivered previously, p discards m; otherwise p proposes m as the next message to be conservatively delivered by the processes of G. When m is conservatively delivered, it is not immediately delivered to the application. Instead, it is inserted in a *consPending* list for later being conservatively delivered, which will happen once every group G' in $sendersTo(G)$ have

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 then G as a destination:

  In this case, when $now > m.ts + w(p)$, 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. Before proposing m, p checks whether m – or a message $m' : m'.ts > m.ts$ – has already been decided in some previous consensus instance. If it has, m is discarded; otherwise, p starts a consensus instance proposing m. Once any message m is decided, if $G \in m.dst$, m is inserted in the *consPending* list. Besides, when m is decided, p sends a message $\{m, \text{'cons'}\}$ to every $p' : (\exists G' \in m.dst \setminus \{G\}) \wedge (p' \in G')$. When $\{m, \text{'cons'}\}$ is received by each $p' \in G'$, p' checks whether it has ever inserted m in its own *consPending* list. If not, p' inserts m into *consPending* and adjusts $barrier(G, G')$ to $m.ts$. To ensure that, once a message $\{m, \text{'cons'}\}$ is received, there is no other message $\{m', \text{'cons'}\} : m'.ts < m.ts$ coming from $m.src$, every $\{m, \text{'cons'}\}$ message is sent through a lossless FIFO channel[4].

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

  In this case, when m is optimistically delivered by a process p of group G, nothing else is done. The message m is inserted into the *consPending* list of p only p receives some message $\{m, \text{'cons'}\}$.

The messages in the *consPending* list are always sorted in ascending order of their timestamps. When the first message m, in the *consPending* 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 "global FIFO total order"[5].

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; $Propose(I, val)$, which proposes a value val for the consensus instance of id $I.id$ within the group $I.grp$; and also $Decide(I, val)$, which is called when the consensus instance I finishes. $Decide(I, val)$ is called for all the processes of the group $I.grp$, when they learn that the value val has been agreed upon in I. 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[6].

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

[5] Proof needed.

[6] This can be easily done by delaying the callback of $Decide(I, val)$ while there is some unfinished consensus instance $I' : I'.grp = I.grp \wedge I'.id < I.id$.

Moreover, each process p of group G keeps an *optPending* list which contains the messages waiting to be *OPT-Delivered* (optimistically delivered). The *consPending* list contains the messages ready to be *CONS-Delivered* (conservatively delivered), but which may be waiting for the clearance from the groups in $sendersTo(G)$. Also, each message m has at least three fields: dst, which is the set of m's destination groups; src, which is the group where m was generated; and ts, the value of the wallclock of the process when m was generated. Note that, by abuse of notation, we have 'process $p \in m.dst$' instead of 'process p : $\exists$ group $G \in m.dst \wedge p \in G$'.

Something that must be noticed is that some messages might not have the source group as a destination. Because of that, to ensure that messages of this kind are also received by $group(p)$'s coordinator c, which then initiates the Paxos instance that allows for the conservative delivery of the message in all of its destinations, every message m from p must also be sent to c (l. 6). This way, when the clock value at the coordinator $time(c) > w(c) + m.ts$, c will conservatively send m to all the processes in $m.dst$. Finally, to avoid delivering a message m in a group it is not addressed to, before OPT-Delivering it, each process must check whether it is one of m's destinations or not (l. 17). In l. 6, m is sent to the whole group G, so that in the case of failure of G's coordinator, it is more likely that the new coordinator will know about the existence of m.

3.1.3 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 conservative delivery. However, it does not disrupt the optimistic delivery. Also, liveness for the conservative delivery 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. For the coordinator of G to know when a message was successfully delivered to other group via consensus, it must also learn it. For this reason, when it proposes any message for conservative delivery, it always includes G as one of the group of learners (l. 16 of Algorithm 1). To avoid the conservative delivery of messages whose destinations do not include G, it checks before including it in the conservative delivery queue (l. 20 of Algorithm 1). This check also avoids delivering empty messages sent by other groups with the sole purpose of providing liveness.

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 is less than $m.ts$ by an infinitesimal difference. This would mean that, if G has no messages to send to G', G' will have to wait for $barrierThreshold + 2\Delta$, assuming that 2Δ is the time needed for a proposal to be learnt after it has been initiated. Because of that, the conservative delivery of m will be delayed by $w(c') + 2\Delta + barrierThreshold + 2\Delta$, where $w(c')$ is the wait window of the coordinator of G', and 2Δ is counted twice because there are two consensus

Algorithm 1 Delivery algorithm, executed by every process p from group G

1: Initialization
2: **for all** $G' \in sendersTo(G)$ **do**
3: $barrier(G', G) = -\infty$

4: *When p generates a message m*
5: $m.ts \leftarrow getTime()$ {*current wallclock value as the timestamp of m*}
6: **for all** $p' \in m.dst \cup \{G\}$ **do**
7: $send(p', m)$ {*send optimistically m to all involved processes*}

8: *When a process p receives a message m'*
9: **if** $m'.ts < getTime() - w(p)$ **then**
10: discard m' {*late commands probably lead to out-of-order delivery*}
11: **else**
12: $optPending \leftarrow optPending \cup \{m'\}$

13: *When* $\exists m \in optPending : getTime() > m.ts + w(p) \wedge \nexists m' \in optPending : m'.ts < m.ts$
14: $optPending \leftarrow optPending \setminus \{m\}$
15: **if** p is a coordinator $\wedge$ $m.src = G$ **then**
16: $Propose(m, m.dst \cup G)$
17: **if** $G \in m.dst$ **then**
18: OPT-Deliver(m)

19: *When $Learn(m)$*
20: **if** $G \in m.dst$ **then**
21: $consPending \leftarrow consPending \cup \{m\}$
22: **if** $m.src \neq G$ **then**
23: $barrier(m.src, G) = m.ts$

24: *When* $\exists m \in consPending : \forall G' \in sendersTo(G) : m.ts < barrier(G', G) \wedge \nexists m' \in consPending : m'.ts < m.ts$
25: $consPending \leftarrow consPending \setminus \{m\}$
26: CONS-Deliver(m)

Algorithm 2 Achieving liveness; executed by the coordinator c of group G

1: Initialization
2: **for all** $G' \in receiversFrom(G)$ **do**
3: $lastBarrierSent(G') = -\infty$

4: *When* $Learn(m) \wedge m.src = G$
5: **for all** $G' \in m.dst$ **do**
6: $lastBarrierSent(G') \leftarrow m.ts$

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

proposals – first in G' regarding m, and then in G regarding the empty message.

There is a way to provide liveness withouth such a possibly big delay, although 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 to 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 in $m.dst \cap receiversFrom(G_B)$, 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 $p' \in m.dst \cup G \cup blockers(m)$. Like in Algorithm 1, the message must be addressed to G, so that G's coordinator starts a consensus instance to deliver it conservatively; and 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 coordinator c' of some group G' receives m, such that $m.src \neq G'$, c' knows that there might be other groups depeding on $G's$ barrier to deliver m. For that reason, it will immediately create a *null* message with a timestamp equals to $m.ts$ and insert it into its optimistic queue. As soon as $null.ts > now - w(c')$, the *null* message will be proposed by c' in G', having $m.dst \cap receiversFrom(G')$ as its learners – 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 learns the *null* message which was just created. The new delivery algorithm would be as described in Algorithm 3.

Now, assuming that $w(c)$ is the same for every group coordinator c, we have a worst-case delivery latency for m of $w(c) + 2\Delta$, as the *null* message is created and proposed in parallel with m. However, each sender process is sending m to every destination process – which will not be changed –, but also to every pro-

Algorithm 3 Delivery algorithm, executed by every process p from group G

1: Initialization
2: **for all** $G' \in sendersTo(G)$ **do**
3: $barrier(G', G) = -\infty$

4: *When p generates a message m*
5: $m.ts \leftarrow getTime()$ {*current wallclock value as the timestamp of m*}
6: **for all** $p' \in m.dst \cup \{G\} \cup blockers(m)$ **do**
7: $send(p', m)$

8: *When a process p receives a message m'*
9: **if** $G \neq m'.src$ **then**
10: $null \leftarrow$ empty message
11: $null.src \leftarrow G$
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** p is a coordinator $\wedge$ $m.src = G$ **then**
22: $Propose(m, m.dst)$
23: **if** $m \neq null$ **then**
24: OPT-Deliver(m)

25: *When* $Learn(m)$
26: **if** $m \neq null$ **then**
27: $consPending \leftarrow consPending \cup \{m\}$
28: **if** $m.src \neq G$ **then**
29: $barrier(m.src, G) = m.ts$

30: *When* $\exists m \in consPending : \forall G' \in sendersTo(G) : m.ts < barrier(G', G) \wedge \nexists m' \in consPending : m'.ts < m.ts$
31: $consPending \leftarrow consPending \setminus \{m\}$
32: CONS-Deliver(m)

cess in its group and to every process belonging to some group of $blockers(m)$. This can create an amount of messages too high to be practical. A solution for that would be the following: instead of sending m to every single process in G and in each group of $blockers(m)$, the sender p sends m only to its destinations and to the coordinators of G and of the groups in $blockers(m)$. That way, the line 6 of Algorithm 3 would be:

for all $p' : p' \in m.dst \vee$
$p' \in \{c : c \text{ coordinates } G \vee (\exists G_B \in blockers(m) \wedge c \text{ coordinates } G_B)\}$
do

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 3.1 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 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[7]. 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.

[7]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.