Practical consistency management for geographically distributed MMOG servers

May 26, 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 $obj(C, S)$ $= \{o : o \in obj(C)$ 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 forwards it to the coordinator. 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.

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.

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 that a message m_1 is delivered before a message m_2, which came from a different source than m_1, if the time t_1, at which m_1 was generated at the process p_1 which sent it, is such that $t_1 < t_2$, where t_2 is the time at which m_2 was generated at p_2, according to p_2's clock, where p_2 is the source of m_2.

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 of its wallclock, which will be $st(m)$, 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 $st(m_i) < st(m_j)$ then m_j is not delivered after m_i in p_d.

3.1.1 Optimistic delivery

The basic idea of the optimistic ordering is the following: assuming that the servers have a synchronized clock[2], whenever a server S (either coordinator or

[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.

[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

replica) receives a message m from a client, it immediately applies a timestamp ts for it, which consists simply of the current value of the server's wallclock, now. Therefore, $m.ts = now$. A wait window of length $w(S)$ is considered, where $w(S)$ is defined as the highest estimated communication delay between the server S and any of the other servers in its group or in its group's neighbors.

After applying the timestamp to m, the server S immediately forwards it to all the other servers involved, including servers in other groups. A server is involved with a message m when it is one of its destinations, in $m.dst$. Then, S puts m on a list (*optPending*), where it stays until $now > m.ts + w(S)$, 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 clients and forwarded by other servers, may have been received and also inserted in that list, always sorting by the timestamp of their timestamps. If $w(S)$ has been correctly estimated as the highest communication delay between the server and each of the other servers in its group or in its neighboring groups, and no message was lost, than all the commands that were supposed to be delivered before m have necessarily been received already. If the same has occurred for all the servers, then all of them have received all the commands, and can deliver them in the same order of ts.

However, even if the optimistic order is the same for all the servers, 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 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 coordinator 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 no more messages with a timestamp lower than t would be sent to group G_{recv}. Let $sendersTo(G) = \{G' : G' \text{ is able to send a message to } G\}$. When a server S, 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 S knows that no

our delivery order prediction succeeds and matches the conservative delivery order.

more messages with timestamp lower than t are coming from other groups and that, once the local ordering 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 server 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.

There are three possibilities for each message C, in the perspective of the coordinator S_c of a given group G:

- Command C affects the state of objects in only one server group.

  In this case, when C is optimistically delivered by the coordinator, it is inserted in a *consPending* list for later being conservatively delivered via consensus in the local group.

- When C affects objects in more than one group and it has originated in the same group G of the coordinator S_c.

  In this case, when C is optimistically delivered by the coordinator, it starts a Paxos consensus instance within its group, having not only its own group's servers as learners, but also those of the neighbor groups affected by C. The proposed value is the command C itself, including $C.ts$. When the value is decided and learnt, the servers of every other affected group G' adjust $barrier(G, G')$ to $C.ts$ and the coordinators of all the involved groups insert it into the respective *consPending* lists.

- When C affects objects in more than one group, but its group of origin is not G.

  In this case, when C is optimistically delivered, nothing else is done by S_c, as it should be inserted into the *consPending* list only when learnt via a consensus instance started at its group of origin – stored in $C.src$.

The list *consPending* is always sorted in ascending order of the timestamp of the commands in it. When the first command C in this list is such that $C.ts < now - w(S_c)$ and $C.ts < barrier(G, G')$ for all G' such that G' is a neighbor of G, then C is atomically broadcast (e.g. also with Paxos) to the group as the next command to be conservatively delivered.

The possibility not covered here is when a message is sent from a group to another one which is not its neighbour. However, it is realistic for this context to assume that objects should be close to each other in the virtual environment to interact and that, as such, this kind of interaction is not supported. Nevertheless, one simple way to provide this kind of support could be by each group waiting for the barriers of every other group in the system, instead of only those of its neighbors.

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; $Propose(val, learners)$, which initiates a Paxos [**?**]

instance proposing the value *val* with the local group as the group of acceptors and *learners* as the group of learners; and also $Learn(val)$, which is called when a paxos instance achieves consensus. The $Learn(val)$ is called for all the processes which are supposed to learn a given proposed value once it is decided by a Paxos instance. For the sake of simplicity, we assume that, for each group, the values are learnt by consensus in the same order in which the consensus instances were initiated. This can be easily done with a sequence number given by the group's coordinator when initiating each consensus instance – but note that we don't care if consensus instances initiated by different groups are finished in a non-deterministic order, but only those initiated in each single group; we want something like a FIFO "learn order", and not necessarily atomic.

Moreover, each server contains an *optPending* list which contains the commands waiting to be *OPT-Delivered* (optimistically delivered). The *consPending* list contains the commands ready to be *CONS-Delivered* (conservatively delivered), but which may be waiting for the clearance from the neighbor groups. Also, each command *cmd* has at least three fields: *dst*, which is the list of groups which contain objects affected by it; *src*, which is the group to which the client who issued it is connected; and *ts*, the value of the wallclock of the server when it received the command from the client. Note that, by abuse of notation, we have 'server $s \in C.dst$' instead of 'server $s : \exists$ group $G \in C.dst \wedge s \in G$.

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 and liveness for the conservative delivery can be easily provided by sending periodic empty commands to neighbors to which no message has been sent for a specified time threshold *barrierThreshold*. Algorithm 2 describes this.

3.1.3 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 optimistic delivery order will match the conservative one. When it doesn't, it is considered a *mistake*. Every mistake of the optimistic algorithm – 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

Algorithm 1 Delivery algorithm

```
1: This part of the algorithm is executed by every server $s$:
2:
3: When $s$ receives a command $cmd$ from a client
4:   $cmd.ts \leftarrow getTime()$   {current wallclock value as the timestamp of cmd}
5:   for all $s' \in cmd.dst$ do
6:     $send(s', cmd)$   {send optimistically cmd to all involved servers}
7:   $optPending \leftarrow optPending \cup \{cmd\}$
8:
9: When a server $s$ receives a command $cmd'$ from another server
10:   if $cmd'.ts < getTime() - w(s)$ then
11:     discard $cmd'$   {late commands probably lead to out-of-order delivery}
12:   else
13:     $optPending \leftarrow optPending \cup \{cmd'\}$
14:
15: When $\exists c \in optPending : getTime() > c.ts + w(s) \wedge$
    $\nexists c' \in optPending : c'.ts < c.ts$
16:   OPT-Deliver(c)
17:   for all $cli \in clients(s)$ do
18:     $send(cli, \{c, \text{'optimistic'}\})$   {s sends c to its clients, warning it is optimistic}
19:   $optPending \leftarrow optPending \setminus \{c\}$
20:   if $s$ is a coordinator $\wedge$ $|c.dst| = 1$ then
21:     $consPending \leftarrow consPending \cup \{c\}$
22:
23: When $Learn(\{c, \text{'deliver'}\})$
24:   CONS-Deliver(c)
25:   for all $cli \in clients(s)$ do
26:     $send(cli, \{c, \text{'final'}\})$   {s sends c to its clients, telling it is the final order}
27:   if $s$ is a coordinator then
28:     $consPending \leftarrow consPending \setminus \{c\}$
29:
30:
31: The next part is executed only by the coordinator $s_c$ of each group $g$:
32:
33: When $\exists c \in optPending : getTime() > c.ts + w(s_c) \wedge$
    $|c.dst| > 1 \wedge c.src = g \wedge \nexists c' \in optPending : c'.ts < c.ts$
34:   $Propose(\{c, \text{'multicast'}\}, c.dst)$   {saving c in the acceptors before multicasting}
35:
36: When $Learn(\{c, \text{'multicast'}\})$
37:   $consPending \leftarrow consPending \cup \{c\}$
38:   if $c.src \neq g$ then
39:     $barrier(c.src, g) \leftarrow c.ts$
40:
41: When $\exists c \in consPending :$
    $c.ts < getTime() - w(s_c) \wedge$
    $\forall g' : g'$ is a neighbor of $g$, $c.ts < barrier(g', g) \wedge$
    $\nexists c' \in consPending : c'.ts < c.ts$
42:   $Propose(\{c, \text{'deliver'}\}, g)$
```

Algorithm 2 Achieving liveness

```
1: This algorithm is executed only by the coordinator $s_c$ of each group $g$:
2:
3: When $Learn(\{c, \text{‘multicast’}\}) \wedge c.src = g$
4:   for all $g' : g' \in c.dst$ do
5:     $lastBarrierSent(g') \leftarrow c.ts$
6:
7: When $\exists g' : g'$ is a neighbor of $g \wedge$
   $getTime() - lastBarrierSent(g') > barrierThreshold$
8:   $ec \leftarrow$ empty command
9:   $ec.ts \leftarrow getTime()$
10:  $ec.src \leftarrow g$
11:  $Propose(\{ec, \text{‘multicast’}\}, \{g, g'\})$        {saving that nothing was sent until $ec.ts$}
```

deliverd out of order, and it is removed from the list – if Q_{opt} doesn't contain C_c, it was probably lost[3]. 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.

[3]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.