Practical consistency management for geographically distributed MMOG servers

May 3, 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 wor '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 replicas, instead of the 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 is 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 may send its commands to any of his server's replicas – which then forwards it to the server. 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, considering that each group consists of a server and its replicas. 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. There are three cases to consider:

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 in-order atomic delivery. For this reason, we also use an *optimistic* delivery for the commands of the players. This delivery, although not guaranteeing that all replicas will receive all commands, or that they will be delivered in the same order, is designed to have a fairly low latency, counting from when each command 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 roll-backs would be necessary. To explain how this works, we first should understand the idea behind the optimistic delivery.

3.1 Optimistic delivery

The basic idea of the optimistic ordering is the following: assuming that the servers have a synchronized clock[1], whenever a server S (either coordinator or replica) receives a command C from a client, it immediately applies a provisory (optimistic) timestamp pTS for it, which consists simply of the current value of the server's wallclock, now. Therefore, $C.pTS = now$. 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 provisory timestamp to C, the server S immediately forwards it to all the other servers involved, including servers in other groups. A server is involved with a command C when one of the objects in $obj(C)$ is coordinated by that server's group. Then, S puts C on a list, where it stays until $now > C.pTS + w(S)$, which means that C has been in that list for a time longer than the defined wait window. In the meantime, other commands, sent from other clients and forwarded by other servers, may have been received and also inserted in that list, always sorting by the provisory timestamp of these commands. 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 C 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 pTS.

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.

3.2 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 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 command affecting the state of objects in more than one group. So, all involved groups must somehow agree regarding the delivery order of these commands. However, as in most MMOGs objects interact only with other objects which are nearby, we can exploit the

[1]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 enought, so that our delivery order prediction succeeds and matches the conservative delivery order.

fact that the groups addressed by a single command are usually neighbors. We do this by defining *barriers* for multicast, such that $barrier(G_{send}, G_{recv}) = t$ means that the group G_{send} promised that no commands with timestamp lower than t would be sent to group G_{recv} anymore. When a server S has received all the barrier values from its neighbours, and they are all greater than a value t, then S knows that no more commands are coming from other groups and that, once the local ordering is done, all the commands 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, it means that it knows all the messages sent until the time value stored in that barrier.

There are three possibilities for each command 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 $C.pTS$. When the value is decided and learnt, the servers of each of the other groups G' adjusts $barrier(G, G')$ to $C.pTS$ 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 $C.pTS < now - w(S_c)$ and $C.pTS < barrier(G, G')$ 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, in a 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.

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 pTS, 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 $G \in C.dst \wedge s \in G$.

3.3 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 deliverd out of order, and it is removed from the list – if Q_{opt} doesn't contain C_c, it was probably lost[2]. Then, the optimistic state is overwritten with the

[2]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

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: $cmd.pTS \leftarrow getTime()$ {*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** $cmd'.pTS < getTime() - w(s)$ **then**
11: discard cmd' {*late commands probably lead to out-of-order delivery*}
12: **else**
13: $optPending \leftarrow optPending \cup \{cmd'\}$
14:
15: *When* $\exists c \in optPending : getTime() > c.pTS + w(s) \wedge \nexists c' \in optPending : c'.pTS < c.pTS$
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* $\exists c \in optPending : getTime() > c.pTS + w(s_c) \wedge |c.dst| > 1 \wedge c.src = g \wedge \nexists c' \in optPending : c'.pTS < c.pTS$
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: $barrier(c.src, g) \leftarrow c.pTS$
40:
41: *When* $\exists c \in consPending :$
$c.pTS < getTime() - w(s_c) \wedge$
$\forall g' : g'$ *is a neighbor of* g, $c.pTS < barrier(g', g) \wedge$
$\nexists c' \in consPending : c'.pTS < c.pTS$
42: $Propose(\{c, \text{'deliver'}\}, g)$

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.

optimistically delivered, the algorithm will know that it should only discard that command, instead of updating the optimistic state.