Practical consistency management for geographically distributed MMOG servers

April 8, 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*.

Each player sends its commands to one of the servers[1] – 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 Baseline 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 assigning each object a logical clock. If the timestamp of a command C is smaller than the logical clock of the object it affects, C is not executed against the object; otherwise, C is executed and its timestamp is used to set the logical clock of the objects if affects. 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:

- Command C affects the state of only one game object.

  In this case, upon receiving C, the server forwards C to the object coordinator, which, in turn, locks[2] the object and starts an instance of Paxos in its replication group to assign a timestamp to C. When that run of Paxos finishes, a timestamp TS is assigned to C, the object is unlocked and each server of the group delivers C and executes it as soon as C's timestamp is the smallest one affecting the referred object.

- Command C affects two or more objects, all of which are managed by the same server group.

  The server that received C then forwards it to the group coordinator. In the command, it is informed which objects are affected by it. With this information, then, the coordinator locks all the involved objects, picks the highest value (TS) among their logical clocks and assigns it as a timestamp for C. After that, an instance of Paxos is run in order to store the

[1] We consider that each server is replicated and that each player may send his command to one of the replicas, instead of the main server, if that provides a lower delay between the issuing of a command and its delivery.

[2] Locking an object here means that no two command proposals can be done in parallel for that object. If, while trying to lock an object, it was already locked, the `lock()` call returns a value informing that it failed to lock.

command and its timestamp in the memory of replicas. Upon reaching consensus, the logical clocks of all the affected objects are, then, set to a value higher than TS and they are all unlocked.

- Command C affects two or more objects and at least two of them are not managed by the same server group.

  `(we have to check whether this really works...)`

  We rely on a variation of Skeen's algorithm to implement this case. The server that receives C forwards C to all the coordinators of the objects affected by C. Each coordinator S then locks all objects in $obj(C, S)$ and proposes a timestamp for C. "To propose" here means to achieve consensus (with Paxos, for example) in the group regarding C's timestamp proposed by that group. The value proposed by a group coordinator when it initiates the consensus instance in its group corresponds to the highest value among the logical clocks of the objects in $obj(C, S)$.

  After a group reaches consensus about its timestamp proposal for C, it sends it to the other groups. When a group with a coordinator S possesses all the timestamp proposals, it picks the highest among them (TS), assigns it as a timestamp to C, and unlocks all objects in $obj(C, S)$. For each group whose proposal was not the highest one, the logical clocks of its object(s) affected by C must be increased to a value higher than TS, which is done by means of consensus in the group.

4 Optimistic protocol

By understanding the abstract description of the baseline protocol of the previous section, it is easy to realize that, even when ordering commands relevant to only one group, a significant number of communication steps is required: player to replica; replica to coordinator; consensus (two communication steps in the case of Paxos, when having the coordinator as the leader, using ballot reservation, each acceptor sending phase 2b messages to all other acceptors and assuming a tolerable number of failures) and, once consensus is achieved, each replica can put the command in the delivery queue and send it to each player. That sums up to five communication steps. On a geographically distributed scenario, this number of communication steps may be prohibitive for the "playability" of a vast number of online games.

In order to mitigate this problem, we propose a protocol for *optimistic ordering and delivery* of the commands, which is supposed to run in parallel with the protocol described in the previous section, which from now on will be called *conservative ordering and delivery*. The optimistic order and delivery are correct if they include all the messages delivered with the conservative one and if they are delivered in the same order. Also, each object has an optimistic state and a conservative state. When a command is conservatively delivered to an object, its conservative state is updated, and this state is certainly consistent

– given the assumptions we have made about the system. Likewise, when a command is optimistically delivered, its optimistic state is updated.

The optimistic ordering and delivery is supposed to be much faster than the conservative one. Assuming that it will be, and if the optimistic delivery order is always correct – which means that it correctly predicted the conservative delivery order –, then the objects will always have a valid state within much fewer communication steps, counting from when each command C was issued to when it is delivered and applied to $obj(C)$.

The basic idea of the optimistic ordering is the following: assuming that the nodes have a synchronized clock[3], whenever a server (either coordinator or replica) receives a command C from a client, it immediately applies a provisory (optimistic) timestamp pTS for it, which consists of the current time now plus the estimated communication delay $dc(S)$ between S and the coordinator. Therefore, $pTS = now + dc(S)$. 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 of its group. We will come back to $dc(S)$ later.

After applying the provisory timestamp to C, the server immediately forwards it to the other servers of that group, and 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, 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 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. That is why we include the value $dc(S)$ in the provisory timestamp of each command forwarded by S. Assuming that the clocks are sufficiently synchronized and that the communication delays have been correctly estimated, then if C was sent by S to the coordinator at the instant t, it will be received at the instant $t + dc(S)$. Thus, when two servers S_1 and S_2 forward the commands C_1 and C_2 to the coordinator and we set their timestamps as $pTS = now + dc(S)$, then the coordinator will receive[4] C_1 before C_2 iff $C_1.pTS < C_2.pTS$. Therefore, given the optimistic assumptions we have made, the conservative delivery order (based on

[3]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.

[4]It's very unlikely that provisory timestamps won't clash, but it is reasonable to assume that, for commands affecting the same objects, and given a timestamp resolution of a few miliseconds, they will probably have sufficiently different timestamps, so that the receival order at the coordinator matches the one predicted by its replicas.

the order in which the coordinator received the commands) will be the same as the optimistic delivery order (based on the provisory timestamp assigned by the replicas).

4.1 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 the object and 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 a command was either lost or was optimistically delivered out of order. We then check whether Q_{opt} contains C_c. If it does, it is removed from that list – if Q_{opt} doesn't contain C_c, it was probably lost, or might be the very unlikely case where the conservative delivery happened before the optimistic one, so 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. 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.