[**Replication in networked games: Overview (Part 1)**](https://0fps.net/2014/02/10/replication-in-networked-games-overview-part-1/)

Posted on [February 10, 2014](https://0fps.net/2014/02/10/replication-in-networked-games-overview-part-1/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/)

It has been a while since I’ve written a post, mostly because I had to work on my thesis proposal for the last few months.  Now that is done and I have a bit of breathing room I can write about one of the problems that has been bouncing around in my head for awhile, which is how to implement browser based networked multiplayer games.

I want to write about this subject because it seems very reasonable that JavaScript based multiplayer browser games will become a very big deal in the near future.  Now that most browsers support [WebWorkers](http://www.w3.org/TR/workers/), [WebGL](http://www.khronos.org/registry/webgl/specs/latest/1.0/) and [WebAudio](http://www.w3.org/TR/webaudio/), it is possible to build efficient games in JavaScript with graphical performance comparable to native applications.  And with of [WebSockets](http://www.w3.org/TR/websockets/) and [WebRTC](http://dev.w3.org/2011/webrtc/editor/webrtc.html) it is possible to get fast realtime networked communication between multiple users.  And finally with  [node.js](http://nodejs.org/), it is possible to run a persistent distributed server for your game while keeping everything in the same programming language.

Still, despite the fact that all of the big pieces of infrastructure are finally in place, there aren’t yet a lot of success stories in the multiplayer HTML 5 space.  Part of the problem is that having all the raw pieces isn’t quite enough by itself, and there is still a lot of low level engineering work necessary to make them all fit together easily.  But even more broadly, networked games are very difficult to implement and there are not many popular articles or tools to help with this process of creating them.  My goal in writing this series of posts is to help correct this situation.  Eventually, I will go into more detail relating to client-server game replication but first I want to try to define the scope of the problem and survey some general approaches.

Overview of networked games

Creating a networked multiplayer game is a much harder task than writing a single player or a [hot-seat](http://en.wikipedia.org/wiki/Hotseat_(multiplayer_mode)) multiplayer game.  In essence, multiplayer networked games are [distributed systems](http://en.wikipedia.org/wiki/Distributed_computing), and almost everything about distributed computing is more difficult and painful than working in a single computer (though [maybe it doesn’t have to be](http://dominictarr.com/post/25083602144/distributed-programming-is-easy)).  Deployment, administration, debugging, and testing are all substantially complicated when done across a network, making the basic workflow more complex and laborious.  There are also conceptually new sorts of problems which are unique to distributed systems, like security and replication, which one never encounters in the single computer world.

Communication

One thing which I deliberately want to avoid discussing in this post is the choice of networking library.  It seems that many posts on game networking become mired in details like [hole](http://en.wikipedia.org/wiki/UDP_hole_punching) [punching](http://en.wikipedia.org/wiki/TCP_hole_punching), choosing between [TCP](http://en.wikipedia.org/wiki/Transmission_Control_Protocol) vs [UDP](http://en.wikipedia.org/wiki/User_Datagram_Protocol), etc.  On the one hand these issues are crucially important, in the same way that the programming language you choose affects your productivity and the performance of your code.  But on the other hand, the nature of these abstractions is that they only shift the constants involved without changing the underlying problem.  For example, [selecting UDP over TCP at best gives a constant factor improvement in latency (assuming constant network parameters)](http://conferences.sigcomm.org/sigcomm/1998/tp/paper25.pdf). In a similar vein, the C programming language gives better realtime performance than a garbage collected language at the expense of forcing the programmer to explicitly free all used memory. However whether one chooses to work in C or Java or use UDP instead of TCP, the problems that need to be solved are essentially the same. So to avoid getting bogged down we won’t worry about the particulars of the communication layer, leaving that choice up to the reader.  Instead, we will model the performance of our communication channels abstractly in terms of [bandwidth](http://en.wikipedia.org/wiki/Bandwidth_(computing)), [latency](http://en.wikipedia.org/wiki/Latency_(engineering)) and the [network topology](http://en.wikipedia.org/wiki/Network_topology) of the collective system.

Administration and security

Similarly, I am not going to spend much time in this series talking about [security](http://en.wikipedia.org/wiki/Network_security). Unlike the choice of communication library though, security is much less easily written off.  So I will say a few words about it before moving on.  In the context of games, the main security concern is to [prevent cheating](http://en.wikipedia.org/wiki/Cheating_in_online_games).  At a high level, there are three ways players cheat in a networked game:

* [Exploits](http://en.wikipedia.org/wiki/Exploit_(computer_security)): Which use bugs in the game logic to directly manipulate the state for the player’s advantage. (eg. [Flight](http://en.wikipedia.org/wiki/Cheating_in_online_games#Fly_hack), [Duping](http://en.wikipedia.org/wiki/Duping_(gaming)), etc.)
* [Information leakage](http://en.wikipedia.org/wiki/Information_leakage): Which snoops on parts of the state that should not be visible to the player. (eg. [MapHacking](http://en.wikipedia.org/wiki/Cheating_in_online_games#Maphacking), [WallHacking](http://en.wikipedia.org/wiki/Cheating_in_online_games#Wallhacking), etc.)
* [Automation](http://en.wikipedia.org/wiki/Video_game_bot): Which uses scripts/helper programs to enhance player performance and repeat trivial tasks. (eg. [AimBot](http://en.wikipedia.org/wiki/Aimbot), [Macros](http://en.wikipedia.org/wiki/Macro_(computer_science)), etc.)

Preventing exploits is generally as “simple” as not writing any bugs.  Beyond generally applying good software development practices, there is really no way to completely rule them out.  While exploits tend to be fairly rare, they [can have devastating consequences in persistent online games](http://web.archive.org/web/20061128103439/http:/news.zdnet.com/2100-1040_22-5829403.html).  So it is often critical to support good development practices with monitoring systems allowing human administrators to identify and stop exploits before they can cause major damage.

Information leakage on the other hand is a more difficult problem to solve.  The impact of information leakage largely depends on the nature of the game and the type of data which is being leaked.  In many cases, exposing positions of occluded objects may not matter a whole lot.  On the other hand, in a real time strategy game revealing the positions and types of hidden units could jeopardize the fairness of the game.  In general, the main strategy for dealing with information leakage is to minimize the amount of state which is replicated to each client.  This is nice as a goal, since it has the added benefit of improving performance (as we shall discuss later), but it may not always be practical.

Finally, preventing automation is the hardest security problem of all.  For totally automated systems, one can use techniques like [CAPTCHAs](http://en.wikipedia.org/wiki/CAPTCHA) or human administration to try to discover which players are actually robots.  However players which use partial automation/augmentation (like aimbots) remain extremely difficult to detect.  In this situation, the only real technological option is to force users to install anti-cheating measures like DRM/spyware and audit the state of their computer for cheat programs. [Unfortunately, these measures are highly intrusive and unpopular amongst users](http://news.bbc.co.uk/2/hi/technology/4385050.stm), and because they ultimately must be run on the user’s machine they are vulnerable to tampering and thus have dubious effectiveness.

Replication

Now that we’ve established a boundary by defining what this series is not about it, we can move on to saying what it is actually about: namely [replication](http://en.wikipedia.org/wiki/Replication_(computing)). The goal of replication is to ensure that all of the players in the game have a consistent model of the game state. Replication is the absolute minimum problem which all networked games have to solve in order to be functional, and all other problems in networked games ultimately follow from it.

The problem of replication was first  studied in the distributed computing literature as a means to increase the fault tolerance of a system and improve its performance.  In this sense video games are a rather atypical distributed system wherein replication is a necessary end unto itself rather than being just a means unto an end.  Because it has priority and because the terminology in the video game literature is wildly inconsistent, I will try to follow the naming conventions from distributed computing when possible.  Where there are multiple or alternate names for some concept I will do my best to try and point them out, but I can not guarantee that I have found all the different vocabulary for these concepts.

Solutions to the replication problem are usually classified into [two basic categories](http://jaksa.wordpress.com/2009/05/01/active-and-passive-replication-in-distributed-systems/), and when applied to video games can be interpreted as follows:

* **Active replication**: Inputs from the players are sent to all players in the network, state is simulated deterministically and independently on each client (also called [lock-step synchronization](http://en.wikipedia.org/wiki/Lockstep_protocol) and [state-machine synchronization](http://en.wikipedia.org/wiki/State_machine_replication))
* **Passive replication**: Inputs from the players (clients) are sent to a single machine (the server) and state updates are broadcast to all players. (also called [primary backup](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.5896), [master-slave replication](http://en.wikipedia.org/wiki/Master-slave_(computers)), and [client-server](http://en.wikipedia.org/wiki/Client%E2%80%93server_model)).

There are also a few intermediate types of replication like [semi-active](http://www.cs.utexas.edu/~lorenzo/corsi/cs380d/papers/p225-chandra.pdf) and [semi-passive](http://ddg.jaist.ac.jp/pub/DSS98.pdf) replication, though we won’t discuss them until later.

Active replication

Active replication is probably the easiest to understand and most obvious method for replication.  Leslie Lamport appears to have been the first to have explicitly written about this approach and gave a detailed analysis (from the perspective of fault tolerance) in 1978:

Lamport, L. (1978) “[Time, clocks and the ordering of events in distributed systems](http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#time-clocks)” Communications of the ACM

That paper, like many of Lamport’s writings is considered a classic in computer science and is worth reading carefully.  The concept presented in the document is more general, and considers arbitrary events which are communicated across a network.  While in principle there is nothing stopping video games from adopting this more general approach, in practice active replication is usually implemented by just broadcasting player inputs.

It is fair to say that active replication is kind of an obvious idea, and was widely implemented in many of the earliest networked simulations.  Many classic video games like [Doom](http://www.idsoftware.com/iphone-games/doom-classic/doomdevelopment.htm), Starcraft and [Duke Nukem 3D](http://en.wikipedia.org/wiki/Duke_Nukem_3D#Multiplayer) relied on active replication.  One of the best writings on the topic from the video game perspective is M. Terrano and P. Bettner’s teardown of Age of Empire’s networking model:

M. Terrano, P. Bettner. (2001) “[1,500 archers on a 28.8](http://www.gamasutra.com/view/feature/3094/1500_archers_on_a_288_network_.php)” Gamasutra

While active replication is clearly a workable solution, it isn’t easy to get right. One of the main drawbacks of active replication is that it is very fragile. This means that all players must be initialized with an identical copy of the state and maintain a complete representation of it at all times (which causes massive information leakage). State updates and events in an actively synchronized system must be perfectly deterministic and implemented identically on all clients. Even the smallest differences in state updates are amplified resulting in catastrophic desynchronization bugs which render the system unplayable.

Desynchronization bugs are often very subtle.  For example, different architectures and compilers may use different floating point rounding strategies resulting in divergent calculations for position updates.  Other common problems include incorrectly initialized data and differences in algorithms like random number generation.  Recovering from desynchronization is difficult.  A common strategy is to simply end the game if the players desynchronize.  Another solution would be to employ some distributed consensus algorithm, like [PAXOS](http://en.wikipedia.org/wiki/Paxos_(computer_science)) or [RAFT](http://raftconsensus.github.io/), though this could increase the overall latency.

Passive replication

Unlike active replication which tries to maintain concurrent simulations on all machines in the network, in passive replication there is a single machine (the server) which is responsible for the entire state.  Players send their inputs directly to the server, which processes them and sends out updates to all of the connected players.

The main advantage of using passive replication is that it is robust to desynchronization and that it is also possible to implement stronger anti-cheating measures.  The cost though is that an enormous burden is placed upon the server.  In a naive implementation, this server could be a single point of failure which jeopardizes the stability of the system.

One way to improve the scalability of the server is to replace it with a cluster, as is described in the following paper:

Funkhouser, T. (1995) “[RING: A client-server system for multi-user virtual environments](http://www.cs.princeton.edu/~funk/symp95.pdf)” Computer Graphics

Today, it is fair to say that the client-server model has come to dominate in online gaming at all scales, including competitive real-time strategy games like [Starcraft 2](http://choongsoo.info/docs/starcraft2.netgames12.pdf), fast paced first person shooters like [Unreal Tournament](https://udn.epicgames.com/Three/NetworkingOverview.html) and even massively multiplayer games like World of Warcraft.

Comparisons

To compare the performance of active versus passive replication, we now analyze their performance on various network topologies. Let n be the total number of players, E be the edges of a connected graph on n vertices.  To every edge (i,j) \in E we assign a weight l_{i,j} which is the latency of the edge in seconds.  In the network we assume that players only communicate with those whom are adjacent in E.  We also assume that players generate data at a rate of b bits/second and that the size of the game state is s.  Given these, we will now calculate the latency and bandwidth requirements of both active and passive replication under the optimal network topology with respect to minimizing latency.

In the case of active replication, the latency is proportional to the [diameter of the network](http://mathworld.wolfram.com/GraphDiameter.html).  This is minimized in the case where the graph is a [complete graph](http://en.wikipedia.org/wiki/Complete_graph) (peer-to-peer) giving total latency of O( \max_{(i,j) \in E} l_{ij} ).  The bandwidth required by active replication over a peer-to-peer network is \Theta(n b) per client, since each client must broadcast to every other client, or \Theta(n^2 b) total.

To analyze the performance of passive replication, let us designate player 0 as the server.  Then the latency of the network is at most twice the round trip time from the slowest player to the server.  This is latency is minimized by a [star topology](http://en.wikipedia.org/wiki/Star_network) with the server at the hub, giving a latency of O( \max_{(0,j) \in E} l_{0j}).  The total bandwidth consumed is \Theta(b + s) per client and \Theta(s n + n b) for the server.

Conclusion

Since each player must be represented in the state, we can conclude that s \in \Omega(n) and if we make the additional reasonable assumption that b is constant, then the total bandwidth costs are identical.  However, if s is significantly larger than s n, then we could conclude that peer-to-peer replication is overall more efficient. However, in practice this is not quite true for several reasons.  First, in passive replication it is not necessary to replicate the entire state each tick, which results in a lower total bandwidth cost.  And second, it is possible for clients to eagerly process inputs locally thus lowering the perceived latency. When applied correctly, these optimizations combined with the fact that it is easier to secure a client-server network against cheating means that it is in practice a preferred option to peer-to-peer networking.

In the next few articles, we will discuss client-server replication for games in more detail and explain how some of these bandwidth and latency optimizations work.