**[Replication in network games: Bandwidth (Part 4)](https://0fps.net/2014/03/09/replication-in-network-games-bandwidth-part-4/)**

Posted on [March 9, 2014](https://0fps.net/2014/03/09/replication-in-network-games-bandwidth-part-4/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/" \o "View all posts by mikolalysenko)

[Last time](https://0fps.net/2014/02/26/replication-in-networked-games-spacetime-consistency-part-3/), I finished up talking about latency issues for networked video games. Today I want to move onto the other major network resource, which is [bandwidth](http://en.wikipedia.org/wiki/Bandwidth_(computing)#Network_bandwidth_capacity). Compared to latency, bandwidth is much easier to discuss. This has been observed many times, perhaps most colorfully by Stuart Cheshire:

S. Cheshire. (1996) “[It’s the latency stupid!](http://rescomp.stanford.edu/~cheshire/rants/Latency.html)“

Though that article is pretty old, many of its points still ring true today. To paraphrase a bit, bandwidth is not as big a deal as latency because:

* Analyzing bandwidth requirements is easy, while latency requirements are closely related to human perception and the nature of the game itself.
* Bandwidth scales cheaply, while physical constraints impose limitations on latency.
* Optimizing for low bandwidth usage is a purely technical problem, while managing high latency requires application specific design tradeoffs.

Costs of bandwidth

But while it is not as tough a problem, optimizing bandwidth is still important for at least three reasons:

1. Some ISPs implement [usage based billing](http://en.wikipedia.org/wiki/Usage_Based_Billing_(UBB)#Usage-based_billing_.28UBB.29) via either traffic billing or download quotas. In this situation, lowering bandwidth reduces the material costs of running the game.
2. High bandwidth consumption can increase the latency, due to higher probability of packets splitting or being dropped.
3. Bandwidth is finite, and if a game pushes more data than the network can handle it will drop connections.

The first of these is not universally applicable, especially for small, locally hosted games. In the United States at least, most home internet services do not have download quotas, so optimizing bandwidth below a certain threshold is wasted effort. On the other hand many mobile and cloud based hosting services charge by the byte, and in these situations it can make good financial sense to reduce bandwidth as much as possible.

The second issue is also important, at least up to a point. If updates from the game do not fit within a single [MTU](http://en.wikipedia.org/wiki/Maximum_transmission_unit), then they will have to be split apart and reassembled. This increases [transmission delay](http://en.wikipedia.org/wiki/Transmission_delay), [propagation delay](http://en.wikipedia.org/wiki/Propagation_delay), and [queuing delay](http://en.wikipedia.org/wiki/Queuing_delay). Again, reducing the size of a state update below the MTU will have no further effect on performance.

It is the third issue though that is the most urgent. If a game generates data at a higher rate than the overall throughput of the connection, then eventually the saturated connection will increase the latency towards infinity. In this situation, the network connection is effectively broken, thus rendering all communication impossible causing players to drop from the game.

Variables affecting bandwidth

Asymptotically the bandwidth consumed by a game is determined by three variables:

* n = number of players
* f = frequency of updates
* s = the size of the game state

In this model, the server needs to send O(s f) bits/(second \* player) or O(n s f) total bits. Clients have a downstream bandwidth of O(s f) and upstream of just O(f). Assuming a shared environment where every player sees the same state, then s \in \Omega(n). This leaves us with two independent parameters, n and f, that can be adjusted to reduce bandwidth.

Player caps

The easiest way to conserve bandwidth is to cap the number of players. For small competitive games, like Quake or Starcraft, this is often than sufficient to solve all bandwidth problems. However, in massively multiplayer games, the purpose of bandwidth reduction is really to increase the maximum number of concurrent players.

Latency/bandwidth tradeoffs

Slowing down the update frequency also reduces bandwidth consumption. The cost is that the delay in processing player inputs may increase by up to O(\frac{1}{f}). Increasing f can help reduce lag but only up to a point, beyond which the round trip time dominates.

Some games use a variable update frequency to throttle back when they are under high load. For example, [EVE Online](http://www.eveonline.com/) does this to maintain stability when many players are clustered within the same zone. They call this technique “time dilation” and explain how it works in a short dev blog:

CCP Veritas. (2011) “[Time Dilation Video Demo](http://community.eveonline.com/news/dev-blogs/2437)“

Decreasing the update frequency causes the game to degrade gracefully rather than face an immediate and catastrophic connection loss due to bandwidth over saturation. We can estimate the amount of time dilation in a region with n players, given that that the throughput of the connection is B bits/second as,

f' \approx \frac{B}{n^2}.

That is the time dilation due to bandwidth constraints should vary as the inverse square of the number of players in some region.

State size reductions

As we just showed, it is always possible to shrink the bandwidth of a game as low as we like by throttling updates or capping the number of players. However, both of these methods weaken the user experience. In this section we consider reductions to the state size itself. These optimizations do not require any sacrifices in gameplay, but are generally more limited in what they can achieve and also more complicated to implement.

Serialization

The most visible target for state bandwidth optimization is message [serialization](http://en.wikipedia.org/wiki/Serialization), or [marshalling](http://en.wikipedia.org/wiki/Marshalling_(computer_science)). While it is tempting to try first applying optimization effort to this problem, serialization improvements give at most a constant factor reduction in bandwidth. At the risk of sounding overly judgmental, I assert that serialization is a [bike shed issue](http://bikeshed.com/). Worrying too much about it up front is probably not a wise use of time.

One approach is to adopt some existing serialization protocol. In JavaScript, the easiest way to serialize objects is to use [JSON.stringify()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify). There are also a number of binary equivalent JSON formats, like [BSON](http://bsonspec.org/) or [MessagePack](http://msgpack.org/) which can be used interchangeably and save a few bytes here and there. The main drawback to JSON is that serialized objects can’t contain any circular references, and that once they are serialized they will lose all of their function properties and prototypes. As a result, it is easier to use JSON for sending small hierarchically structured messages than process gigantic amorphous interconnected networks of objects. For these latter situations, [XML](http://en.wikipedia.org/wiki/XML) may be more suitable. XML has the advantage that it is extremely flexible, but the disadvantage that it is much more complex.

Both JSON and XML are generic, schema-less serialization formats and so they are limited with respect to the types of optimizations they can perform. For example, if a field in an object is a US phone number then serializing it in JSON takes about 9 bytes in UTF-8. But there are only 10^9 possible US phone numbers, and so any such number can be encoded in \log_2(10^9) \approx 30 bits, or a little under 4 bytes. In general, given more type information it is possible to achieve smaller encodings.

Google’s [Protocol Buffers](http://en.wikipedia.org/wiki/Protocol_Buffers) are an example of a serialization format which uses structured schemas. In the XML world there has been some research into [schema based compressors](https://www.ibm.com/developerworks/library/x-datacompression/), though they are not widely used. For JSON, there don’t appear to be many options yet, however, there has been some work on creating a [standard schema specification](http://json-schema.org/) for JSON documents.

Compression

Another way to save a bit of space is to just compress all of the data using some standard algorithm like LZMA or gzip. Assuming a streaming communication protocol, like TCP, then this amounts to piping the data through your compressor of choice, and piping the output from that stream down the wire.

In [node.js](http://nodejs.org/), this is can be implemented with the built in [zlib stream](http://nodejs.org/api/zlib.html) libraries, though there are a few subtle details that need to be considered. The first is that by default zlib buffers up to one chunk before sending a write, which adds a lot of extra latency. To fix this, set the Z\_PARTIAL\_FLUSH option when creating the stream to ensure that packets are serialized in a timely manner. The second and somewhat trickier issue is that zlib will split large packets into multiple chunks, and so the receiver will need to implement some extra logic to combine these packets. Typically this can be done by prefixing each packet with an integer that gives the length of the rest of the message, and then buffering until the transfer is complete.

One draw back to streaming compression is that it does not work with asynchronous messaging or “datagram” protocols like UDP. Because UDP packets can be dropped or shuffled, each message must be compressed independently. Instead, for UDP packets a more pragmatic approach is to use less sophisticated entropy codes like [Huffman trees](http://en.wikipedia.org/wiki/Huffman_coding) or [arithmetic encoding](http://en.wikipedia.org/wiki/Arithmetic_coding). These methods do not save as much space as zlib, but do not require any synchronization.

Patching

Another closely related idea to compression is the concept of patch based updates. Patching, or diff/merge algorithms use the [mutual information](http://en.wikipedia.org/wiki/Mutual_information) between the previously observed states of the game and the transmitted states to reduce the amount of data that is sent. This is more-or-less how [git](http://git-scm.com/) manages commits, which are stored as a sparse set of updates. In video games, patch based replication is sometimes called “[delta compression](http://en.wikipedia.org/wiki/Delta_encoding)“. The technique was popularized by John Carmack, [who used it in Quake 3’s networking code](http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/Quake3Networking).

Area of interest management

Finally, the most effective way to reduce the size of updates is to replicate fewer objects. That is, given any player we only replicate some subset of the state which is relevant to what the player is currently seeing. Under the right conditions, this *asymptotically* reduces bandwidth, possibly below n^2. More generally, updates to objects can be prioritized according to some measure of relevance and adaptively tuned to the use available bandwidth.

These techniques are called *area of interest management*, and they have been researched extensively. At a broad level though, we classify area of interest management strategies into 3 categories:

* Rule based: Which use game specific rules to prioritize updates
* Static: Which split the game into predefined disjoint zones
* Geometric: Which prioritize updates based on the spatial distribution of objects

This is roughly in order of increasing sophistication, with rule based methods being the simplest and geometric algorithms requiring the most analysis.

Rule based methods

Some games naturally enforce area of interest management by the nature of their rules. For example in a strategy game with “[fog of war](http://en.wikipedia.org/wiki/Fog_of_war#Simulations_and_games)“, players should not be allowed to see hidden objects. Therefore, these objects do not need to be replicated thus saving bandwidth. More generally, some properties of objects like the state of a remote player’s inventory or the AI variables associated to computer controlled opponent do not need to be replicated as well. This fine grained partitioning of the state into network relevant and hidden variables reduces the size of state updates and prevents cheating by snooping on the state of the game.

Static partitioning

Another strategy for managing bandwidth is to partition the game. This can be done at varying levels of granularity, ranging from running many instances of the simulation in parallel to smaller scale region based partitioning. In each region, the number of concurrent players can be capped or the time scale can be throttled to give a bound on the total bandwidth. Many older games, like EverQuest, use this strategy to distribute the overall processing load. One drawback from this approach is that if many players try to enter a popular region, then the benefits of partitioning degrade rapidly. To fix this, modern games use a technique called instancing where parallel versions of the same region are created dynamically to ensure an overall even distribution of players.

Geometric algorithms

Geometric area of interest management prioritizes objects based on their configurations. Some of the simplest methods for doing this use things like the Euclidean distance to the player.  More sophisticated techniques take into account situation specific information like the geodesic distance between objects or their visibility. A very accessible survey of these techniques was done by Boulanger et al.:

J.S. Boulanger, J. Kienzle, C. Verbugge. (2006) “[Comparing interest management algorithms for massively multiplayer games](https://www.comp.nus.edu.sg/~cs4344/0607s1/netgames06/s01Conf96_a32.pdf)” Netgames 2006

While locality methods like this are intuitive, distance is not always the best choice for prioritizing updates. In large open worlds, it can often be better to prioritize updates for objects with the greatest visual salience, since discontinuities in the updates of these objects will be most noticeable. This approach was explored in the SiriKata project, which used the projected area of each object to prioritize network replication. A detailed write up is available in Ewen Cheslack-Postava’s dissertation:

E. Cheslack-Postava. (2013) “[Object discovery for virtual worlds](http://ewencp.org/dissertation/object-discovery-for-virtual-worlds/)” PhD Dissertation, Stanford University

Pragmatically, it seems like some mixture of these techniques may be the best option. For large, static objects the area weighted prioritization may be the best solution to avoid visual glitches. However for dynamic, interactive objects the traditional distance based metrics may be more appropriate.

Final thoughts

Hopefully this explains a bit about the what, why and how of bandwidth reduction for networked games. I think that this will be the last post in this series, but probably not the last thing I write about networking.