Chunky: A Dynamically Sharded Distributed Multiplayer Game Framework

Aashish Welling
Shreyas Kapur

All our source code and benchmarking scripts can be found at the following link: https://github.com/omegablitz/chunky.

1 INTRODUCTION

Massively multiplayer game servers typically use a single server to which multiple game clients connect. Scaling and supporting a higher number of players usually requires hardware upgrades in the form of better CPUs.

Sometimes, if the game worlds are larger than what a single server can handle, game developers resort to splitting the world onto multiple servers with interim load screens as the players are relayed from one server to another. This limits player interaction between two world regions and constrains developers to only a few game design choices.

In this project we present Chunky: a distributed, scalable and fault-tolerant system to support large worlds and players, scaling efficiently as the number of servers increase. We present a method to make the server switches seamless, and allow boundary player interactions without constraining the game developer to have fixed world boundaries.

We analyze and build our system on the popular sandbox game, *Minecraft*, for convenience and large community supplied codebases, while making our implementation general for any game. We implement our system and benchmark it against other high performance, single machine based servers on a number of tasks.

2 DESIGN

2.1 Goals

We attempt to design a multiplayer game framework server with the following features in mind:

- **Scalability:** we want to be able to string together multiple machines to create one large seamless world while maintaining performance.
- **Fault-tolerance:** typically when game servers crash, all the data that hasn't been flushed to disk is lost and the players wait for server to come back online. We want to design our system such that in case of server failure, players will be moved to another server with the last flushed data.
- **Ease of use:** we want to provide an easy to use API for game developers to use our game framework as a drop in library.

2.2 Bottlenecks

Each game world provided by the developer can be divided into *chunks*. A chunk is defined as a discretized $c \times c \times y$ section of the game world. In *Minecraft*, the discretized unit is called a *block* and a chunk is a section of the world of $16 \times 16 \times 256$ blocks.

6.824, May 2018,
2018.

At any given time, most games keep certain chunks loaded in server memory. The server also runs the game loop on these loaded chunks. These chunks are typically loaded around players, as the server needs to run the main game logic around them. The number of chunks loaded around each player is known as the *view distance*.

World files for games are usually small enough to be kept on disk. The primary bottleneck arises with keeping chunks and entities loaded in memory and running the game loop on these loaded objects. A single server quickly gets saturated as the number of players increase, and hence the number of loaded chunks.

2.3 Invariants

In a traditional world-splitting approach, player-player interaction between shared world boundaries would not be possible without delicate state synchronization across the two servers. If state-sharing is used, it incurs bandwidth costs if the states are large (in the case of Minecraft) and has potential for worlds to diverge states.

Instead, chunky doesn't split the world into static regions; instead, it assigns *ownership* of specific chunks to servers. Our system maintains three key invariants:

- A chunk can at most be owned by one server.
- Players are always connected to servers that own the chunk that the player is located on.
- Chunks of players within the same view-distance of each other are owned by the same server.

These invariants allow us to keep players that can potentially interact with each other on the same server, thereby allowing player-player interaction critical for any game.

2.4 Operation

Figure 1 shows the system overview diagram. Chunky uses a single manager as an overseer for operations, a flexible amount of stateless proxies, the game servers, and a shared network file system.

The servers never save anything to the shared world file unless explicitly asked to by the manager. This allows us to neatly handle world load-save races.

During normal operation, the game client connects to one of our proxies (at random, load balanced) which gets the server the player should join from the manager. The proxy then forward all traffic from the client onto the specified game server.

The manager periodically asks each of the game servers for a list of chunks that are loaded in memory as a result of the players on the server. The loaded chunks are typically all chunks in view distance of each player, but can also be based more on special conditions like merge radius etc. Based on these loaded chunks, the manager maintains a list of *blobs*: a set of contiguous chunks. These messages also serve as heartbeats, making the manager aware of which chunk servers are online and ready to receive players.

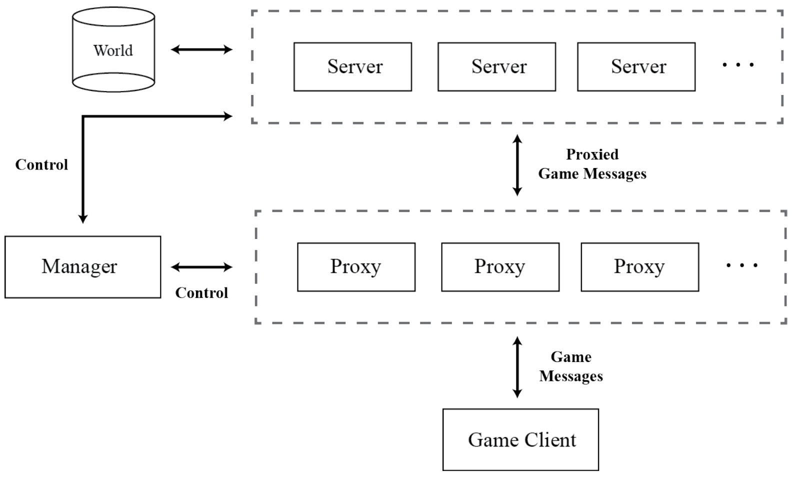

Figure 1: An overview of Chunky: the manager controls operations, and proxies relay game messages to the correct game server which run the main game loop. The world files are shared over the network to all game servers and are never written unless the manager asks to write game states to the disk.

The manager attempts to maintain Invariant 1 & 3. First, it assigns each blob to a single server, minimizing transitions and distributing all server resources appropriately (discussed later in Section 2.5). The manager then calculates all chunk transitions, i.e. chunks that have a different owner than before, and sends a *disk flush* RPC to servers that own those chunks. This instructs the game servers to write the specified chunk states to the shared disk. The manager then asks the new owners to load these chunk states from the shared file.

The manager then attempts to maintain Invariant 2, calculating if players are on servers that don't own the chunks they're located on, instructing the proxies to switch servers for such players.

Periodically, the manager reaches out to all servers and asks each server to flush its owned chunks to disks, akin to a world auto-save on most games.

2.5 Blob Assignment

While there exist efficient solutions to dynamically maintain connected components on a graph, the manager runs an elementary depth first search to calculate chunk blobs. We found that this was efficient enough for our benchmarking purposes.

There are three cases when servers need to be reassigned ownership:

- **Blob Combine:** when two blobs become a single blob, we make a vote of whoever already owns the majority of chunks in the new blob, and assign chunk ownership of all chunks within the blob to the winning server. An alternate strategy could be to minimize the number of player switches, the tradeoff being player-switches vs state transfer costs.
- **Blob Split:** when two blobs split, we assign one of the blobs to the server that has the least number of chunks loaded if its load is significantly less. Otherwise, we keep the two blobs on the same server.
- **New Blob:** a new blob is assigned to the server that has the least number of chunks loaded.

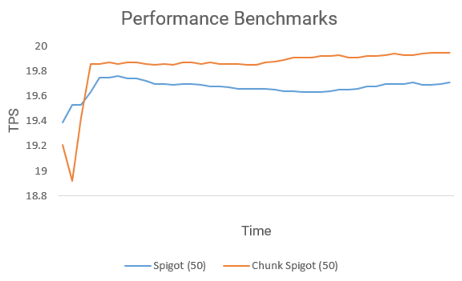

Figure 2: Our benchmark results on both Spigot and Chunky synchronized in time with $n = 50$ and $b = 1000$. We realized that setting b to a small number means that all players are on the same server, which defeats the purpose of Chunky. We discuss more on this in Section 4.

2.6 Fault-Tolerance

In case a game server goes offline and the manager doesn't receive the list of loaded chunks from the server, the manager attempts to distribute chunks previously owned by the failed server onto other servers.

In the process, it swaps players onto the new servers, giving the illusion that the fail never happened, and increasing uptime.

The new game state may be a stale version. This is still better than existing solutions, as with existing solutions we would lose both data and uptime. Chunky attempts to always be up, even if the data is a few seconds stale.

3 EVALUATION

Figure 2 shows our benchmark results. We test our platform on a benchmark we developed. Our benchmark consists of spawning n benchmarking players on the server, spreading them b blocks away from each other and making the players perform random tasks like moving and placing blocks. Across all tests, we kept the random generator seeds constant and the same initial world states for consistency.

The performance of *Minecraft* servers are typically measured in TPS (higher the better) which is the number of ticks the server can simulate every second. The TPS (as in this case) is usually capped at 20.0.

We compared Chunky against *Spigot*, a state-of-the-art Minecraft server written for performance. Spigot runs on a single CPU core. For this benchmark, Chunky uses only two Spigot server backends. The tests were run on the same machine, simulating data-center level latencies. For each test, we collected TPS over time, synchronizing, running the test 3 times for both Chunky and Spigot and taking the max TPS.

4 FUTURE WORK

As of now, our system performs and distributes well for sparse player locations, but starts to break if players are densely packed. While we relax the condition that dense player regions are unlikely, we think it's possible to resolve this issue, at least partially by state sharing under certain conditions.

Another drawback of our system is that sparse chains of people invariably make all the chunks loaded on the same server. This again is a relaxed assumption, but can be resolved by some level of state sharing across boundaries when the number of players on a single server grows.

We also need to evaluate the scalability of our system as the number of players and servers increase. Due to the time constraint, we omitted this evaluation.

4.1 Conclusions

Chunky is a novel system for distributing workloads that are characteristic of multiplayer games. Previously, game developers would need to choose between having a small world with full immersion and having a large world with disconcerting transitions when switching in between these world. Chunky allows for the creation of large worlds that are completely immersive, and in addition fault-tolerant and automatically rebalancing.