conclusion.org

Tomas Herman

2012-04-18 Wed

Contents

1 Conclusion **1**
- 1.1 Comparison to official server . . . 2
- 1.2 Testing . . . 2
- 1.3 Review of design . . . 3
- 1.4 Review of used tools . . . 3
  - 1.4.1 Scala . . . 3
  - 1.4.2 Akka . . . 4
  - 1.4.3 Redis and Netty . . . 4
- 1.5 Room for improvement and new features . . . 5

1 Conclusion

In the last part of this thesis, I will try to compare my implementation of server with official implementation, I will talk about how I tested the project specification. I will try to review and judge my decisions made during the design phase of the project. I will review the tools I used and talk about how well did they performed for the task. I will also propose new features and improvements to be implemented in the future. And lastly, I will try to summarize all the interesting stuff I learned during this project.

1.1 Comparison to official server

Unfortunately, official implementation of the Minecraft server is not open sourced and is obfuscated, so there is not too much information available. We can still compare the two in a few aspects, though.

It is known, that official implementation uses file system as storage of the map fragments. My implementation uses Redis database, which stores data in memory and only flushes them to disk after certain period of time. While the locally stored map has it's advantages, such as speed and simplicity, it would be very hard to create distributed server using such approach because we would need to either synchronize or split the map chunks to different servers. Synchronization would add a lot of additional traffic and complexity while splitting chunks would be very fault intolerant. If one server would have failed, entire part of map would become unavailable. Also, it would be very hard to coordinate events that happened on the edges where the map would have been split. Imagine an explosion - event which affects blocks in a radius from epicenter. If it happened on the edge of the map, we would not only need to update blocks on the part of the map where the explosion was triggered, we would also need to notify the neighbor server about event.

With Redis, we get the map synchronization for free. Redis can work in a cluster (experimental feature for now) and from users point of view, we just write into a single node instance, but in the background Redis will automatically update all the instances in the cluster.

A great advantage of Specus over official implementation is the design with extensions in mind. While there is unofficial and successful Bukkit project [1] which aims to provide API for plugin creation for the official server, I can only imagine how hard people had to work to reverse engineer official server in order to provide such api. On the other hand, entire Minecraft is implemented as plugin in Specus and thanks to the design of the plugin architecture, user extensions can not only add their own packets and behaviors, but also hook callbacks on packets from any other plugin and thus allowing extensions to cooperate with each other.

1.2 Testing

Testing was quite a big problem during this project. Obviously, I was able to use common techniques of testing, such as unit testing and integration testing during the development of Specus, but testing of complete server could not be automated and had to be done by hand.

As one might suspect, there is no command line client for Minecraft (that I am aware of) that would allow for some sort of automated testing. So I would have to write my own client in order to test it properly, which would by itself probably took as much time as the entire server implementation.

[1] www.bukkit.org

Another fact that made testing hard was the fact that Minecraft is paid game and I owned only one copy. Minecraft is also quite resource heavy. On my desktop machine, I almost ran out of memory with on a very lightweight system (ArchLinux with XMonad desktop environment, which by itself uses only about 4% of memory) while having 1 copy of Minecraft client running, 1 server instance, 1 node instance, 1 instance of Redis database, IntelliJ IDE and Simple build tool[2] so testing with multiple client instances would be very inconvenient or almost impossible with the machinery I had available.

So the actual testing was done using my experience and knowledge of what the server was supposed to do. While not very clean or academical, it was unfortunately only possible solution considering the time constraints.

1.3 Review of design

Minecraft itself is still under heavy development and it's creators don't really seem to care about breaking backward compatibility and don't mind introducing new packet types, modifying old ones or even adding or removing new data types. While that was a little annoying, it gave me a chance to test the flexibility of the designed architecture.

I am happy to say, that I think i did a good job with the architecture design. For example, when a format of `LoginPacket` was changed in a patch, all I had to do was to update the packet definition in Minecraft plugin and code handling the packet and I was done. Smart codec described in the Design and Implementation part of the paper took care of all the low level encoding and decoding.

1.4 Review of used tools

1.4.1 Scala

I have to say, I am very happy I chose Scala as programming language for this project. While there were some downsides to it which I will address below, the overall experience was very pleasant.

Thanks to the functional style of coding, I didn't manage to find almost any bugs in most of the code during unit testing. That is, in my opinion, due to the fact that in functional programming one writes a lot of functions that focus on one thing only, with no side effects. That kind of code is easy to reason about and easy to get right.

[2]Build tool for Scala projects.

Unfortunately, I managed to run into a compiler bug once which compiled source code into a byte code that would throw `InitializationException` upon invocation. I wasn't able to find the reason for the exception so I had to rewrite code in different fashion.

I got a chance to test how well Scala works with libraries designed for Java when using Netty library. I had no problems using it. The code looks comparable to Scala code.

1.4.2 Akka

Akka is a very impressive piece of software. The only problem thing i don't like about the way they implemented the Actors is that user looses a great deal of type safety. Any Actor can be accessed only through `ActorRef`, which gives no indication of the type of an Actor.

It would be nice if there was some way to determine the instance of an actor or at least be able to check what types of messages can Actor processes. The reason it can't be done in Akka is the fact that Akka actors can dynamically change their behavior and change which and how the messages are processed.

Other than that, I had no problems with Akka. I used more concepts from the framework for example I used `TransactionalMap` to track mapping between `SessionID` and Netty Channels. `TransactionalMap` is basically a persistent immutable map which also implements interface of mutable map. It uses `AtomicRef` to store map internally and guarantees that the `update` method is atomic and can be safely called from multiple threads at once.

I also used `Future` objects, which take a function and execute it in different thread. They have very useful API, which allows user to use execute a number of different `Futures` and then invoke different function when all other functions are done. This is used for example when streaming the map chunks to player for the first time. We create requests for sending the map in a future, then we wait until they are all finished and then we send player the instruction to spawn.

1.4.3 Redis and Netty

I had no problems using Redis nor Netty. I must say I was very impressed with the simplicity of both of theirs API. Netty especially provides a very easy to use API which doesn't bother user with the low level implementation of networking.

1.5 Room for improvement and new features

Of course, there is plenty of work to be done in order to improve the current implementation.

As far as the new features go, I would like to see web admin implemented using the HttpFrontned plugin. Also finishing the Minecraft implementation would be desired.

One of the more interesting thing that would be nice to implement would be a DSL for Redis communication, that would abstract away the fact that the entire communication is done using `Future` monads. As of now, most of the Minecraft node is plagued with `map` and `flatMap` calls.