full-clean.org

Tomas Herman

2012-05-04 Fri

Contents

1 Thanks

I would like to thank to the following people and communities for help with my thesis:

- Mr. Miroslav Uller for valuable comments and guidance
- Everyone from Minecraft Coalition Wiki [1] for all their hard work on writing great documentation of Minecarft protocol
- Everyone from #mcdevs IRC channel for help with debugging and understanding of Minecraft protocol
- Mr. Victor Klang and everyone from Typesafe for creating Akka framework and for all the help they provided to me via mailing list

[1]http://www.wiki.vg/Main$_{\text{Page}}$

2 Introduction

It seems like more and more these days, people rely on remote services and application which store data or even entire business logic on a server, while providing only a thin client for the user to interact with. That puts a lot of responsibility on the creators of such applications to create a quality, highly available service, that can be trusted to work correctly.

In my thesis I try to capture my hands on experience with writing one such server using very modern techniques and tools in order to create architecture, that is easy to reason about, maintain and extend. I will be implementing a subset of business logic for Minecraft game, which should be demanding enough to prove the points I'm making further in the paper.

3 Minecraft

In this chapter I will provide a brief overview of what Minecraft is and some of it's most important concepts. Hopefully, after reading this chapter, reader will agree that Minecraft is quite interesting game with a lot of potential.

3.1 What is Minecraft

3.1.1 Overview

Minecraft is a indie game developed by Markus Presson and Mojang which was published in 2011. It's a open world game, in which players are placed in a world made of blocks. These blocks can be mined and used as a building material.

Players use these blocks to build various items or structures. For example, there has been successful attempts to build USS Enterprise, Taj Mahal, Eiffel Tower and basically anything one could imagine.

3.1.2 Maps

When a game of Minecraft is started, player may choose to generate a new map. This is done using map generator, with random initial seed (player may also choose to use specific seed). When a game starts, Minecraft server generates a small area around player. As the player moves around the map, more and more parts of the map are generated when needed.

Maps in Minecraft are made of blocks. To organize these blocks, maps are split into so called *chunks*. These chunks are simply 16 blocks wide, 16 blocks deep and 128 blocks high [2]. When a client connects to server, server sends the map using these chunks.

Maps can be very large. There is a hard limit of 256 block on the height of the map. The depth and length of the map, however, has a soft limit of 2^{32} (because of limits of integers in Java) (if player moves further then that, the map chunks starts to get overwritten). Which means that there can be up to 2^{2*32+8} blocks per map. Even if every block was represented by 1 byte, it can be quite challenge to represent such large object efficiently.

3.1.3 Blocks and Items

There is a number of different blocks in Minecraft. Among the most common ones there is dirt, stone, sand and gravel. There are also trees, which can be broken down into wood. There are also rare blocks, such as coal, iron, gold and diamond. Each of these blocks has various properties and uses.

When mining these, player can use either a bare hands or craft an item that would aid him. There is a lot of items player can craft in Minecraft, but the basic ones are axe, shovel, hoe and pickaxe. Each of these can be made from either wood, stone, gold, iron or diamond which determines it's quality. Items are crafted by placing blocks into 2x2 or 3x3 matrix into different shapes, which determines the item to be crafted.

Tools described above are useful for increasing efficiency of gathering blocks. For example iron pickaxe can crack stone a lot faster then wooden pickaxe. Some of the blocks can't even be gathered without good enough tool. Diamond block, for instance, require at least iron pickaxe.

There are other items player can craft, though. For example, player can create a furnace, which uses coal blocks as fuel and can smelt ores into bar as well as cook food from raw meat that can be gathered from sheep, cows or pigs. It can also "cook" sand block into glass or cobble stone into smooth stone.

Minecraft also supports alternative forms of transportation with boats or mine carts, which can be placed upon a rails in order to move faster between locations. Rails make use of so called *Redstone energy*, which is described below.

In order to protect himself, player can create and equip armor and weapons in either leather, gold, iron or diamond quality. There are 4 pieces of armor: helmet, chest piece, trousers and boots. As weapons there is only sword and bow and arrows. Armor reduces damage taken from monsters

[2]http://notch.tumblr.com/post/3746989361/terrain-generation-part-1

while weapons increase players damage to monsters and fauna of Minecraft worlds.

3.1.4 Monsters & health

Every player has 10 hearts that symbolize his health. Every heart can be either full, half empty(also known as half full) or empty. When all hearts are empty, player dies and is either re spawned, or in case the player plays in *hardcore mode* the entire world is deleted and all game content is lost.

There is also a food counter, which represents how well fed the player is. If the bar is full, player automatically regenerates health if he has not taken any damage in recent history. This is to prevent health regeneration while fighting enemies.

Plenty of opportunities to loose health are implemented in Minecraft. Player looses health when dropping from high enough edge, while being under water for too long or while standing in fire or lava.

The most common cause of health loss, however, are monsters. Monsters spawn in the places where there is no light available. Light can come either from sun, torches, fire or lava. There is a number of monsters in Minecraft:

- Zombie
  Slow melee monster that deals quite a lot of damage, when killed drops meat that can be cooked and eaten.

- Skeleton
  Shoots arrows, when killed drops arrows or bones.

- Creeper
  Very quiet monster which creeps up on player and explodes when in proximity of player. Makes sizzling noise before detonation.

- Spider
  Melee creature, which only attacks player during night or when attacked.

- Silverfish
  Melee creature spawning from blocks which look exactly the same as stone blocks in randomly generated fortresses and dungeons.

- Enderman
  Melee creature that can teleport, but attacks player only if player looks at it first. Otherwise it's not hostile.

3.1.5 Redstone

Redstone is one of the most interesting features of Minecraft. Redstone is a rare ore that can be found deep in the ground. When mined, it produces several Redstone crystals.

These crystals can be either used for crafting, or laid on other blocks. Player can use these to create kind of a wire made of the Redstone crystals. The wiring acts like a carrier of logical values. By default, the value transmitted by the wire is 0. It can be changed, though. In Minecraft community, this is usually called *Redstone energy* and the state in which logical 0 is transmitted via wire is considered as lack of *Redstone energy*.

There is a couple of ways how to send logical 1 via Redstone wire:

Item	Description
Redstone torch	Sends 1 permanently
Button	Sends 1 as impulse
Leaver	Sends 1 as long as the leaver is triggered
Pressure pad	Sends 1 as long as something is on the pad

There are also items that can 'consume' Redstone wire in order to perform action (non exhaustive table):

Item	Action
Door	Open while 1 is transmitted
TNT	Triggers explosion once 1 is transmitted
Note block	Emits sound once per 0 to 1 value change
Dispenser	Dispenses object once per 0 to 1 value change

Using these tools, Minecraft users were able to create some very impressive structures. There is for example a calculator implementation, song playing machines or even games created with Redstone infrastructure.

Other than that, Redstone circuits are often used in *adventure maps* for creating puzzles and challenges. Typical example of Redstone usage would be asking player to find a button, in order to open doors into next part of the map. It can also be used for creating traps, by wiring TNT or dispensers with arrows to pressure pads.

3.1.6 Nether

Nether is an alternative map which is available to players via portals. Portal is a 5 blocks high and 4 blocks wide frame with 3 blocks high and 2 blocks wide space inside made of obsidian, which is lit using flint and tinder. Obsidian is a block that is created by pouring water over lava blocks.

It symbolizes kind of an evil realm with some unique resources, but overall is not overly interesting. The portal system, however, is used quite often in *adventure maps*.

3.1.7 Goals of the game

Minecraft is very open ended game, so there is no real ending to the game. The only formal ending to the game requires player to find one of many randomly generated underground fortresses, build a portal inside and go through. There the player will find a dragon, which he must slay. However, once that is done and credits have passed, the game still can be played.

More often than not, though, players don't even bother with this quest and play the game only for the joy of building interesting structures. Game usually ends when player gets bored. Unless player plays on hardcore mode, which automatically deletes the world upon players first death.

3.1.8 Creative mode

Creative mode was added to Minecraft in order to make it easier for people to create impressive structures. Those people may not want to necessarily deal with all the stuff Minecraft contains, such as monsters, inventory management, mining blocks and so on.

In creative mode, player has access to infinite resources from within his inventory. He can also destroy any block with 1 hit and is allowed to fly. He also takes no damage and spawning of monsters is disabled.

Player may choose to play in creative mode when starting a new game. There are extensions, however, that allow player to switch creative mode on and off at will.

3.1.9 Adventure maps

One of the reasons Minecraft got so popular are maps made by players, which usually contain a story, quests and riddles for player to go through.

Adventure map is a regular map, which usually contains additional document which describes the story, rules of the map (usually forbids player to destroy any blocks) etc. Adventure maps heavily utilize the use of redstone wiring for creating "scripted" events.

3.2 Extensions

As one might imagine, Minecraft would be a very good platform and engine to build on. Unfortunately, there is no API for players to build upon. Players still managed to reverse engineered the code, though, in order to create plugins and extensions for the game. And they really managed to make some amazing plugins. In this part, I will mention few of the most interesting extensions.

3.2.1 Tekkit mod

Tekkit is a collection of multiple extensions, which adds concept of the electrical power to the game (among other things). It adds randomly generated pools of oil into the maps, which can be gathered, processed to fuel and used in electrical engines to power machines. There is a lot of machines that consume electricity, but the most interesting one is a quarry, which automatically mines selected area.

There are other ways to gather electricity, though. There are for example nuclear reactors, which players can build. They need to be cooled down, however, or they will explode and contaminate area with radioactivity.

3.2.2 Computercraft

Another very interesting extension is Computercraft. It adds programmable robots into the game. Robots are programmed via in game terminal using embedded LUA [3] programming language (added by the extension).

There are for example mining probes, which can be programmed to search for given materials, mine only those and return them to the owner. It can also be used for password protecting doors. It can be even used to implement an text-based RPG (in game terminals are text-only and Computercraft doesn't include any tools for creating graphical UI).

3.2.3 Other extensions

There is a great number of extensions. Just to quickly mention a few others, there is an extension that adds mini map for players. There is an extension that adds gps-like navigation and ability to create points of interest. There are extensions that add new items, enemies or blocks and so on.

[3] www.lua.org

4 Goals

In this part of the paper I will talk about goals of the project I will be working on. In the first part, I will reason about why I chose the goals the way I chose them, while in the second part I will provide a brief summary of the goals in form of a list.

4.1 Reasoning

I wanted to make this project a learning experience, which affected a lot of the decisions about which technologies to use as well as what subset of functionality described in the part about Minecraft to implemented.

Reader would hopefully agree that while Minecraft is based on quite simple ideas, it is still a complex universe with a lot of details to implement. I wanted to focus mainly on basics, which I thought at the time would be most important for further development in the future.

4.1.1 Server related goals

The main focus of this project is the server infrastructure, which I hoped would be very independent of Minecraft itself. If I would have had done my work correctly, Specus (that is how i named the project, it means 'cave' in Latin, which i thought was appropriate) server could be used for any other game or project easily.

- Simplicity
  I wanted the server infrastructure to be very simple to use and simple to reason about, because as I learned in my previous projects, building concurrent systems with networking IO can be quite difficult to get right. In order to achieve that, I used Scala programming language, which is said to have great support for concurrent programming.

  I also decided to use Actor pattern, which seemed very interesting and very natural to use when dealing with concurrency.

  I also wanted to abstract away all the IO operations and the lower level mechanics of the server. I didn't want to deal with no buffers, sockets or channels when working on business logic.

- Extensibility
  From the description of Minecraft above, I hope it is clear to the reader how important, fun and interesting are the Minecraft extensions. That

is why I wanted my server to be built with extensions in mind from the start.

I wanted it's extension system to be powerful enough to be able to implement entire Minecraft business logic as extension (extensions are called Plugins later in the text and in the code).

I wanted plugin programmers to be able to express dependencies on other plugins, as it's very common case that a plugin wants to extend or cooperate with functionality provided by other plugins.

- Distributivity
  I wanted my server to be able to spread the workload into multiple machines, because Minecraft it self has quite big problems with the workload. As mentioned above, map can contain up to $2^{(32+32+8)}$ blocks, so I felt it was important to be able to save all these data into remote database (or cluster of databases).

4.1.2 Minecraft related goals

Because I felt like I chose quite ambitious goals for the server architecture, I decided to keep it simple with the actual logic implementation and treat the Minecraft business logic as a proof of concept. I decided, for now,to only implement just the creative mode described above. That allowed me to skip the implementation of inventory management and monsters, which would take a lot of time.

I also decided to not implement any complicated map generator. I implemented a very simple one for testing purposes which generates simple flat stone world.

I decided not to implement in game maps, signs and items that required any special handling.

I wanted to implement map streaming and on-the-fly map generation, map updates when player makes a change and persistent player position (position of a player is persisted between sessions).

While that is not very impressive set of features, it should provide and test all the important features of the server architecture.

4.2 List summary of goals

Following is the brief summary of the goals mentioned above in form of a list:

- server architecture requirements

  - implemented in Scala
  - extensive usage of Actor model
  - extensible via plugins
    * must be able to express dependencies on given plugin and it's version
    * must be powerful enough to be able to express entire Minecraft logic
  - IO and socket networking abstracted away
  - state moved from local variables into remote database

- Minecraft functionality requirements
  - player position persistence
  - on the fly map generation
  - map streaming
  - map updates by player
  - implemented as plugin
  - must store all the state in a remote database

5 Scala

In this chapter, I will try to explain why i chose to use Scala language for implementing Specus. Discuss strengths of Scala compared to other alternatives and provide a quick overview of the most useful features which I used in Specus. In the last part of this chapter, I will discuss some of the weaknesses of Scala and describe how i used the features mentioned in the rest of this chapter.

5.1 Why Scala?

There were several reasons which lead me to choose Scala for Specus implementation. First of all, I wanted to make this project a learning experience. And ever since i took Haskell/Lisp class, I was interested in functional programing. I think that functional programming will become more popular and more desired skill to have in years to come, due to the increasing demand on correct and concurrent software. I am also quite experienced with Java language. Scala provides very good support for functional programming while still preserving many concepts from object oriented programming. So it seemed like a natural choice to choose Scala.

5.2 JVM

Scala source is compiled into JVM bytecode. That means that any Scala projects automatically benefits from all the effort people have put into optimizing JVM aswell as features that speed up computations during runtime (JIT compilation, code inlining etc). JVM programs are, obviously, platform independent (as long as Java Runtime Environment is avalible for given platform), so one gets platform independence for free.

There also exist a number of great and mature tools and libraries written and compiled for JVM platform, which can be very easily used while working with Scala. For example one could use a Proguard[4] program to minimize the jar produced by Scala compiler by removing the unused classes from libraries and compiled code.

5.3 Quick Scala overview

Scala was designed by Martin Odersky and his team at ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE [fn::http://www.epfl.ch/index.en.html]. The name stands for Scalable language, which describes the language rather well. Please note, that *Scalable language* is not meant in a sense of horizontal/vertical scalability (Scala is as good as any language in that sense of a the word), but authors rather meant it in a sense that the language features scale with the experience of user [fn::http://www.scala-lang.org/node/8610] . In Scala, it is relatively easy to design libraries that appear to be language features. For example the new *try with resource* statement added in JDK7 [5] could be implemented in Scala on library level very easily.

Scala is a rather unique mix of object oriented concepts and functional programming concepts with very powerful standard library, which contains, among other things:

- rich collection framework with both mutable and persistent implementations
- parallel collections (collections, whose methods are processed in multiple threads)
- parser combinators (library for simple writing of powerful parsers)
- wrappers around many of JDK features for more Scala-like usage

[4]http://proguard.sourceforge.net/

[5]http://docs.oracle.com/Javase/7/docs/technotes/guides/language/try-with-resources.html

5.4 Object oriented features

Much like in Java, code in Scala is organized using constructs from object oriented programing. In Scala, there are 3 basic entities: Classes, Objects and Traits. Following is the brief overview of each of the entities.

5.4.1 Traits

In Scala, trait are kind of an mix between Java interfaces and abstract classes. Traits can define method, which can either be left abstract (trait only defines the header of the method, implementation is left to the user) or can contain implementation as well. Traits can not only define methods, but fields as well, although it's recommended to use methods, which can be later overwritten by vals.

Traits can extend 0 or more traits. Trait can also declare it's dependency on other entity. For example, we can have a trait `ChatSocket` with method `pullChatData()` which returns array of bytes and we want to create trait `ChatFormatter` with method `printableChatData()` functionality which uses `pullChatData()` and creates formatted string. That means we need to make sure that both of these traits are mixed into same object. We could define the traits like so:

```
trait ChatSocket { def pullChatData(): Array[Byte] = ... }
trait ChatFormatter { self: ChatSocket => printableChatData()
= ... }
```

Now whenever we create object which extends `ChatFormatter`, we need to also extend `ChatSocket` or the code will not compile.

5.4.2 Objects

In Scala there is entity called objects, which is basically a class that is guaranteed to be only presented once in a JVM. It's Java equivalent would be class that is created using Singleton pattern.

Objects can extend traits, but nothing can extend objects. Objects don't have constructors. Every method on object is "static", which is why objects are commonly used as what is called "companion objects" to classes. Companion objects usually contain factory methods as well as other useful functions for given class.

5.4.3 Classes

Classes are very similar to classes from languages like Java. They have constructors, can extend a class and implement 0 or more traits.

5.5 Functional and exotic features

Unlike in Java, Scala supports a vast set of features usually available in functional languages, as well as other useful concepts. Here I will briefly introduce some of the interesting concepts and at the end of the chapter I will try to show example of an interesting application of these concepts.

5.5.1 Pattern matching

One of the features I liked most about Haskell was pattern matching. One can think of pattern matching as about more powerful version of switch/case statements.

User defines a sequence of patterns and callbacks that is called when pattern matches. Patterns are tried in order in which they were defined.

Scala implements this feature by using entities called extractors. Extractors are functions named `unapply`, that are applied to input and return either `Some(value)` or `None`. If the extractor return `Some`, it is considered to match the input. Otherwise the next extractor is tried.

5.5.2 Vars and Vals

Scala has two types of fields: vals and vars. Vals are fields that are guaranteed to be assigned only once and never changed. Vars on the other hand can be changed just like a regular Java variable. It is considered good practice to always use vals, unless it's necessary to use var.

5.5.3 First class functions

In Scala, functions are first class citizens. That means, in Scala one can treat functions like any other datatype. Function can be stored in variable, it can be passed around and created on demand. Functions can return new functions and so on.

Scala compiler creates a Java class for every first class function (methods of objects are created as regular Java methods of objects), so basically storing and passing function becomes simply storing and passing of a reference to the created object. This created class has an `apply([argument-list])` method generated, which represents the function invocation. In Scala there is a syntactic sugar for invoking `apply([argument-list])` methods by simply calling `([argument-list])` on the object. For example `a([argument-list])` is translated to `a.apply([argument-list])`. This means that it's really easy to even create objects/classes that can be used as functions, by simply defining `apply([argument-list])` method.

5.5.4 Case classes

Case classes are quite interesting feature of Scala. They are defined using `case` keyword, like so: `case class X([constructor-arguments])`. For example, lets say we want to create a class representing a point in 3D space. Case class could look like so: `case class Point3d(x: Int, y: Int, z:Int)`.

For such class, Scala compiler will generate a few very useful methods. First of all, a reasonable `toString`, `equals` and `hashCode` methods are generated, which use constructor parameters to compare equality and to generate hash code. A companion objects with factory method and extractor methods are generated for given case class as well. Compiler also generates methods that allow user to access the fields in order they were declared in constructor. This might not seem like a very interesting feature but it is used to great success in Specus and is described below.

It's important to note that constructor parameters of case classes can be accessed (as fields) and are immutable.

5.5.5 Collection API

Scala has very impressive set of collections. It has common data structures - list, vector, stack, queue, map, set and possibly even more. All of these are available in multiple versions. When not specified otherwise, data structures are available as so called "persistent data structures". Persistent data structure is a data structure, that when altered creates what seems like a new instance of data structure with altered content. Original instance remains unchanged. Operations on persistent data structures use clever tricks and structure sharing in order to achieve similar complexities as their mutable versions.

Scala also have mutable versions of data structures. Those are the equivalents of data structures that can be found in most languages.

Very interesting feature of Scala standard library are parallel data structures. Those are persistent data structures, but the interesting thing about them is that methods defined on them like `filter`, `map` etc are executed using multiple threads.

5.6 Weaknesses

As with most tools, there are trade offs when using Scala. In this part of the paper I will talk about some of the negatives I encountered when using Scala.

First of all, Scala is quite a new technology, so the tool support is not as advanced as for example for Java, but it is getting better. I used IntelliJ Idea IDE with Scala plugin when developing Specus and it was reasonably pleasant experience. It supports basic refactoring, code completion as well as error highlighting. However, it sometimes reports error in a code that is perfectly correct.

The more important issue with Scala is the naming of all the generated code by Scala compiler. It can sometimes be difficult to figure out when and why exceptions are being thrown, especially because it's common to use so called "one liners" quite often when dealing with collections and so on, which condense quite a lot of logic into 1 line of code and are usually littered with anonymous functions. Every time we use anonymous function, Scala compiler generates a class representing that function and gives it some generic name. It uses the classpath to package in which the function is defined followed by `$` followed by some arbitrary text to guarantee uniqueness of the name. For example for function in val `f` in object `o` defined like so:

```
object o { val f = () => throw new Exception() }
```

and invoked:

```
o.f()
```

will return following stack trace:

```
Java.lang.Exception at o$$anonfun$1.apply(<console>:7) at o$$anonfun$1.apply(<console
```

One can see how the stack traces could get very unreadable very fast. Luckily, after a while I didn't find this to be a big deal but it was definitely a challenge early on.

5.7 Example usage of case classes and first class functions

In this part of the paper I will talk about what I thought was quite interesting usage of the features described above. First I will explain what I was trying to build and why and then I will go into details of implementation.

Minecraft clients communicate with server using TCP connection. There are about 70 different types of "packets" (by packets i mean logical packets, as TCP is stream service so there are no real packets visible to user) that are being sent over the wire. There are many different ways to implement such mechanism, but the way I chose to do it is to create a case class for every different kind of packet which would represent the fields of packet and a codec, which knows how to take the instance of given packet and encode it into a byte array which can be sent via TCP and read by client. It also knows how to read a byte array and parse it into the given packet case class.

Most naive, but in some languages the only solution would be simply creating codecs by hand and copy-pasting the encoding code in. One might think that it would be possible to use Java reflection API [fn::http://docs.oracle.com/Javase/1.4.2/docs/api/Java/lang/reflect/package-summary.html] to figure out what the type of value are the fields of given packet and parse/encode them accordingly.

And that does work fine for parsing - Java reflection gives us the tools to obtain constructor of given class. From that constructor, we can figure out all it's parameters as well as their types (we can get class object of the parameters) and it gives us a method to programmatically invoke the constructor with array of `Object` values that are used as constructor parameters. Thus providing us with enough power to create generic parser that would figure out how to parse packet just from it's constructor.

The real problem is with encoding the packet. While we can get all declared fields of given class, those fields are given in no particular order [fn::http://docs.oracle.com/Javase/1.3/docs/api/Java/lang/Class.html]. We could of course use tricks like annotations to establish the order of fields, but that would introduce more boilerplate and in the end would make our code more confusing.

Luckily, like described above, case classes provide API for users to access constructor fields in order in which they were defined.

So now we have a way to get types of constructor parameters of given class and we know how to access those fields in order they were defined in. All we need now is some kind of mapping between type of class and a function that would be able to parse and encode that type. But that should be easy, because as described above, functions are first class entities. We can simply create `Map` from `Class` object to `(_ <: Any, ChannelBuffer) => Unit` for encoding (function that takes anything and channel buffer, into which we encode the packet and returns nothing) and `Map` from `Class` to `(ChannelBuffer) => Any` (function that takes channel buffer and returns anything) for decoding.

Above solution has a problem, still. It operates with Any, which basically means we loose all type safety, For example we could put into our map mapping from class of Int to function that returns String. We can't make the type constraints on map any stronger, because we couldn't add all the data types into it, obviously. What we can do is create an API which would use Scala generics and made sure that functions have proper headers and add it to our maps for us.

```
def addType[A](encode:(A)=>ChannelBuffer,decode:(ChannelBuffer)=>A)
```

The method signature above symbolizes how such API could look like. The method takes two functions, one called `encode` which takes argument of type `A` and returns `ChannelBuffer` and function called decode, which takes `ChannelBuffer` and returns type `A`.

Basically, using approach described above, I was able to save myself writing about 60 classes full of boilerplate code, in which it would be very easy to make errors. I still had to implement some codecs by hand, as Minecraft API is not designed very well, though.

6 Actor model

In this chapter, I will discuss why one should care about concurrency, I will take a look at conventional models of concurrent computations on Java Virtual Machine (JVM) and problems that goes along with them. Then I will talk about fundamental concepts of Actor model followed by more detailed description of Akka - my toolkit of choice for actor systems on JVM platform.

6.1 The free lunch is over

"The free lunch is over" is an article written by Herb Sutter that appeared in Dr.Dobb's journal in 2005 [fn::http://www.gotw.ca/publications/concurrency-ddj.htm]. He talks about the end of an era, in which software is getting faster (not more performant) simply by the fact that the hardware in getting faster. He argues, that while historically companies like AMD or Intel focused on increasing the clock speed of CPUs, it is no longer possible, due to physical limitations. So instead what these companies are doing in order to increase power of their products is adding more cores onto the chips.

That means, that in order to harness the power of this new hardware, we need to approach the craft of writing software in a different way. We need to focus on concurrency and we need to focus on creating tools that will make writing concurrent software easier.

6.2 Problems with conventional models of concurrency

Probably the most common concurrency entity used today in programming are threads. Concept of thread comes from operating systems and kind of leaks through into programming language libraries. Thread allows us to execute concurrently with very little (programming) effort. For example all we need to do in Java programming language, is to create instance of class extending `java.lang.Thread` and implement the -public void run()- method. Threads are very convenient that way.

However, there are some very important drawbacks of doing concurrency this way.

6.2.1 Threads are expensive

Because of the way threads work, there is non-trivial amount of work to be done when thread is created. A stack has to be allocated for every new thread (default size is 512kB on JVM) and a number of system calls needs to be made (JVM uses platform specific threads). Generally, creating new threads is considered expensive.

What this means, is that one shouldn't create threads dynamically, every time a concurrent execution is required. Common approach instead is creating a number of threads ahead of time and reusing them (this pattern is sometimes called thread pool). While this is reasonable option, this add a nontrivial complexity to the application and basically means that threads don't scale (we are limited by the number of threads in thread pool).

6.2.2 Thread based concurrency is hard

Threads can be used to a reasonable level of success in some programs. Especially programs that use threads for processing operations that don't need to communicate between each other nor share same resources(for example web servers, build tools etc). Threads then serve as sort of a cheaper processes.

However when dealing with shared resources and shared state, threads become really hard to use. Because threads share memory heap, it is very hard to keep data consistency and because threads can use all the resources on the heap, deadlocks can occur very easily and it is not a trivial exercise to eliminate all the bugs that can come from such model[6].

[6]http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf

6.3 Actor model overview

Actor model is a model of computation, designed to deal with problems in a highly concurrent, asynchronous and fault tolerant fashion. It was first published by Carl Hewitt in 1973 [7]. Actor model is widely used in systems where reliability, availability, scalability and concurrency are important features.And as the number of cores per processors continue to increase, it is reasonable to expect that the demand for tools that promise easier handling of concurrency will increase as well.

Probably the most popular actor implementation today - the Erlang OTP framework, has been used in many software projects and services. Here is a few examples of Erlang applications:

- nosql databases: CouchDB, Riak
- message queues: RabbitMQ
- web servers: YAWS

6.4 Fundamental concepts

In actor model, computation is processed using Actors. By Actor we mean an entity which can:

- send asynchroneous messages to other actors (sender doesn't wait on reply from the receiver)
- receive messages from other actors
- create new actors
- change it's behavior dynamically

Every actor has an inbox, into which system queues messages sent to given actor. Actor processes messages one at a time. When thinking about actor, it helps to imagine it as a kind of lightweight thread (all actors in the system run at the same time), which is very cheap to maintain, create and destroy.

Computation is then split into series of operations that are executed by different actors. Results of those operations are then sent around via messages. It is important to note that there can be many instances of given

[7] Carl Hewitt; Peter Bishop and Richard Steiger (1973). A Universal Modular Actor Formalism for Artificial Intelligence. IJCAI.

actor type. It is therefore important to design system in such a way that actors don't affect each other (for example by holding locks).

For example, let's say we want to create a service that writes logging data into a log file. We could easily create a function in every actor that opens a file, appends the log message and closes the file. That would be problematic, though, because multiple actors might want to write at the same time. We could use locks, to make sure that only 1 write is being issued at a time. However, that would be very inefficient, because essentially only 1 actor in the system would be allowed to run at a time, while other actors would wait for the resource to become available. What we could do instead, is make another actor (lets call it logger), that would hold the reference to our log file and every time an actor would write into the log file, it would send the log message into the logger, which would handle the actual write. Please not that there is no need for locking with this approach. Even if two actors try to write at the same time, it only means that two messages are sent, and actor model guarantees that messages are processed sequentially and only 1 message is processed at a time.

This approach might look similar to object oriented programming, where we create a wrapper around a resource to encapsulate the details of the implementation (such as locking). But it's important to remember, that the messages in Actor model are asynchronous. Which means that actor just sends the message and doesn't wait on response, it just keeps working.

Another important property of actors is that they are very cheap to create (In Akka, overhead for creating an actor is only about 600 bytes). This allows system to generate actors when needed, for example, we could have a web server, that generates a new actor on demand for every incoming connection.

6.5 Enter Akka

Every actor model implementation is different from others. For example, just for Scala programming languages there are 4 different implementations as far as i know (Lift actors, Scala Actors from standard library, Scalaz actors and Akka actors). I decided to use Akka actors, because they come as part of a great library and support remote communication between JVMs, which saved me a lot of work. Akka actors also support Erlang-like fault tolerance and -ask- kind of messaging, which is described below.

6.5.1 Actors

In Akka, actor can simply be created by extending Actor trait and implementing a receive method. In this method user maps different kind of messages to functions for processing given message. It's important to note, that Akka actors don't support any kind of scanning of the inbox (some implementations allow for example checking the length of inbox etc.)

Inside every actor, a self variable is present, containing the important information about state of an actor. For example, one can obtain ActorRef(described below) to sender actor during message processing.

We can then instantiate the actor by calling the factory method `actorOf`. By calling this method, the user only get instance of class `ActorRef`. That instance represents the actor in the system, but does not contain the actor. This is so that the state of the actor can never be compromised, because user can never get reference to the actual actor. `ActorRef` supports methods ! (pronounced bang) and ? (pronounced ask).

The bang method represent a simple 'fire and forget' kind of messaging, while the ask method creates a `Future` object, which has hooks into which user can insert callback methods, which are called when the `Future` is completed. This approach eliminates the need of blocking and waiting until the receiving actor reads and responses to our message.

The `ActorRef` instance is completely thread safe, can be passed around in messages and can even be serialized and sent via network to different JVM and will still refer to the original actor.

6.5.2 Remote access

Akka also supports remote actors. Thanks to the properties of `ActorRef` described above, one can run Akka systems in multiple JVMs and simply by sending `ActorRef` around one is able to communicate with remote actors using the standard actor semantics (! and ? methods).

Akka actors can also be registered by string name in so called "actor repository", from which one can withdraw them remotely. For example in Specus, there is an actor registered in the server under name that is know to nodes. What that means, is that when a node is booted up, it can get a reference to the registered server actor and begin communication.

6.5.3 Fault tolerance

In order to achieve fault tolerance, a supervision scheme is implemented. Conventional programming methodology deals with error using 'defensive programming'. Basically, programmer is trying to check input data for all possible inconsistencies and only when all tests pass, data are allowed to be further processed.

On the other hand, Akka accepts the fact that no code is bug-less, so instead of trying to catch all the invalid cases, it encourages programmers to embrace the failure, and focus on recovery from failure. Every time an exception is thrown in Actor, it gets restarted. By that it's meant that the new, fresh instance of Actor is created and injected into system in such a manner that all the ActorRefs to the original actor are valid and point to the newly created actor. Actor can implement life cycle methods like preRestart and postRestart in order to do save it's state and do anything that needs to be done. The message causing the failure is not processed again, however rest of the mailbox with unprocessed messages is reused for the new actor.

In addition to that, Actors can be assigned into tree-like structures where every node can have at most 1 supervisor and can supervise 0 or more actors. When actor is about to fail and is being restarted, a message is sent to the supervisor, so that it can decide what to do. It can decide whether he wants to restart just the failing actor, or all the actors he oversees (it can sometimes be useful).

7 Design and implementation

In this chapter, I will discuss the relevant information about Minecraft and it's architecture needed in order to write a server. Then I will describe design choices I took when designing Specus and talk about libraries and technologies I used for implementation of Specus.

7.1 Minecraft

Minecraft uses client - server architecture for multiplayer support. 1 client can be connected to only 1 server. Minecraft clients communicate with server using TCP protocol. Data are formatted into logical packets. There is a number of different packet formats. Every packet is prefixed with unsigned byte which indicates the type of packet, which ultimately determines how the rest of stream should be parsed.

Because Minecraft is still being developed, there are usually some changes in protocol and packet types when versions change. At the time of writing this thesis, Minecraft version is 1.1.

7.1.1 Data types

For the most part, Minecraft packets consists of only few well defined data types. Some packets however use ad-hoc formatted data structures. Following is the list of packet type commonly used in protocol:

- integer fields - signed numbers using two's complement encoding
  - byte: 1 byte long, -128 to 127
  - short: 2 bytes long, -32768 to 32767
  - int: 4 bytes long, -2147483648 to 2147483647
  - long: 8 bytes long, -9223372036854775808 to 9223372036854775807
- decimal number fields
  - float: 4 bytes long, range compatible with Java float
  - double: 8 bytes long, range compatible with Java double
- string field: UCS-2 encoded string, prefixed with short (as described above) which signalizes the length of the string
- metadata field: described below

7.1.2 Metadata field

Metadata is a format introduced by Minecraft in order to efficiently (space wise) encode and decode triplets of data (identifier of piece of data, data type and value itself) of variable length. Every triplet begins with a byte. Top 3 bits (with `0xE0` mask) of the byte encode the data type of value while the bottom 5 bits (mask `0x1F`) encode the id of entity. The value itself depends on the data type and is parsed accordingly. If the byte value is 127 (`0xFF`), it means that there are no more data in metadata. The type of data that can be stored in metadata are:

top bits	datatype
0x000	byte
0x001	short
0x010	int
0x011	float
0x100	string
0x101	short, byte, short
0x110	int, int, int

7.2 Design of Specus

When designing Specus, I focused mostly on flexibility and extensibility. It should also be possible to distribute the workload on multiple computers. Minecraft client is built to be connected to 1 server. So i decided to split Specus into multiple parts. There is the server, which is the only part of the Specus that clients can see and there are worker nodes, which are the parts of the system that do the actual work.

7.2.1 API and implementation

Because Specus was designed to be very extensible, it was important to split both server and node projects into two. API and actual implementation. API contains all the stuff that needed to be available for plugins while implementation contains the mechanisms that are not useful to plugins. Also, because node and server communicate together, i decided to create another project, called `common api`, which contains classes that are needed by both.

It contains for example plugin system API (described below), it contains metadata format, it contains `Packet` super class that all packets need to extend and so on.

7.2.2 Server

Only job of server is to accept new clients, read and parse data into Packet case classes and sent them to nodes. It also knows how to encode Packet data from case classes and write them into TCP connection. Process of parsing and encoding is further described in chapter about Scala. It also knows how to send a message to any given node and it accepts messages from nodes.

If a new clients connects to the server, a new unique id is generated for the client. That id is only thing any other component of the system needs to know in order to be write to the client connection. The generated id is valid until the connection closed or server shutdown, whichever happens to happen first.

When TCP data arrive on the server, first byte is read. It is then checked, whether any codec is registered for given byte (remember, every packet type is prefixed with id byte). If a codec is found, rest of the received data is given to that codec for parsing and new instance of packet message is received by chosen codec. That message is then sent, along with id of a client, to one of the connected nodes.

When one of the nodes wants to write a packet message to client, it simply sends `WriteRequest` message, which contains id of client and instance of packet and server will handle the writing for them. Therefore, nodes doesn't need to know anything about actual parsing or encoding packets, which makes it a lot easier to implement nodes, as it only deals with regular Scala (case) classes.

If a connection is closed, server simply removes the id of client from it's internals and sends a notification to a random node, so that it can clean up after the user.

7.2.3 Node

All the actual business logic is done in nodes. Nodes are independent JVMs running node code and are connected using remote actors described in the 'Actor' part of the paper. When a node machine is started, a message is sent to the server upon which server adds the node to the set of available nodes and starts sending messages to it.

In nodes, message processing should be done either in stateless fashion, or the state should be persisted in some sort of database as the messages are sent randomly to the nodes. In order to do that, I use Redis database (which is described below).

Node plugins can contain so called `processors`. Processor is a class that can consume a packet and somehow process it. Each packet can be processed by multiple processors and they are not processed in any particular order. Processors also need to be able to provide a sequence of all the packets they are able to process so that packets are only sent to the processors that actually know how to use them, thus reducing the overhead compared to scenario where all packets are sent to all processes.

7.3 Plugin system

In this subsection I will talk about general design of plugin system implemented in Specus, then I will overview all the implementation details. At the end I will talk about 3 different plugins I implemented as proof-of-concept.

7.3.1 General design

As described above, Specus aims for maximal extensibility. That's why it has been designed to be very plugin friendly from the very beginning. By itself, Specus contains only basic functionality related to generic packet parsing, plugin loading and communication between server and nodes. Everything else is implemented in plugins, including entire Minecraft logic.

Much like Specus itself, plugins too are meant to be separated into two parts. Server part and node part. In server part, plugins can declare packets and codecs for those packets. They can also register for receiving different kinds of messages which indicate what events are happening in the system.

On the other hand, node part of the plugin usually consists of an Actor, or system of Actors, that are registered for different types of Packets parsed by server part of plugin.

Plugin is basically just a jar file which contains a plugin descriptor on predefined class path. Plugin descriptor is a simple file containing a JSON encoded information about plugin, such as it's dependencies, it's version, plugin identifier (string representation of plugin, usually same as the Java package in which the code of plugin is placed), author of the plugin and most importantly the entry point class. It is kept inside the jar file in order to make the handling of plugins as simple as possible. Plugin is expected to communicate with the system using messages (as described in Actor part of this paper).

Entry point class contains additional information required for running the plugin. By default, it can contain a entry point Actor class, which is instantiated when the plugin is loaded and into which the system messages are sent. It can also contain a list of classes, which the particular plugin is interested in. Only those messages would be sent to Actor. Because the plugin API is designed to be reused in both server and node, user of the API can define contents of entry point class as she wishes. For example, in server sub project the entry point contains list of packets and codecs for packets. On the other hand in node code the entry point contains processors for packets.

It is important to note that all the plugin jars must be added to classpath when the user of plugin API is started.

7.3.2 Implementation

In common API, there is abstract class `SimplePluginManager` through which all the plugin loading is done. It contains method `bootupPlugins` which takes a `File`, which represents directory containing plugins. First it attempts to parse plugin descriptor from each .jar file it finds in plugin directory. If everything goes well, we now have a set of all plugin descriptors which contain plugin version and it's dependencies, which means we can now validate that all dependencies are either fulfilled or there is something missing.

Once all the dependencies are checked, entry point classes are instantiated and all the entry point Actors are created and registered for messages they are interested in (as defined in entry point class).

After that a user defined `postDependencyCheck` method is created, which could do anything that needs to be done. For example, in node part of Specus, this is where Minecraft maps are generated in advance. In server part this is where we can for example sent dependencies to the plugin (if plugin needs some). When this method returns, plugins are considered ready for work. If at any point an error occurs, whole server shuts down as it makes no sense trying to recover from these errors.

7.3.3 Communication among plugins

It is very important for a plugin system to support very easy communication among plugins. That is because a plugin can use functionality already implemented by others.

As mentioned above, in plugin descriptor there is a field specifying plugin identifier. This is used to obtain reference from `Plugin Manager`, which is passed to plugin during initialization phase. `Plugin Manager` should always have the correct reference available, as the plugin system already verified that all plugin dependencies are available at this point. The received reference is simple `ActorRef`, as specified in Actor part of this paper, which allows user to simply send messages to it.

7.3.4 Stats

Stats was first plugin I implemented in Specus. I needed a way to track connected users when debugging the server and later I added a feature that collected all the packets sent and received by server per client.

The way it is implemented is quite simple. System broadcasts messages when a new client is {dis,}connected and when a packet is sent or received. Stats plugin waits for these messages and updates it's state accordingly. It contains a counter of connected clients and a map containing list of all sent and received classes of packets per user. This map is a immutable persistent data structure, so when other plugin ask for this data, it can be very efficiently sent (basically it just sends a reference to the map) to it without worrying about someone mutating it and thus destroying the consistency of data.

7.3.5 Http frontend

For a while, the `println` approach of displaying information from stats plugin was ok. But i decided later i needed something more readable. So i created Http frontend plugin. It's only purpose is to display information gathered by stats plugin.

It uses Jetty embedded http server which listens on 9090 port. When a new http request is issued to that port for / resource, it sends message to the stats plugin for most up-to-date data and returns them formatted for easier reading. It should go without saying that this kind of display is a lot easier to read than looking for text in log files of the server.

Also, the plan was to make full featured administration interface using this plugin, which is very possible, but due to time constraints I was not able to implement this feature.

7.3.6 Minecraft

And last but not at all least, the Minecraft plugin. This plugin contains everything that is specific to Minecraft. The plugin itself is split into 3 parts:

- Common API
  Common API is the part of the project that defines all the different packets that can be sent or received by client. It was required to put these into separate jar, so that they can be easily reused. Also, if there was another plugin that would want to enhance functionality of Minecraft plugin, or simply just invoke it's own action when some of the packets defined by Minecraft is received, this would be the jar to use.

- Server
  In a server part of the plugin, there are definitions of all the codecs for each packet defined in common api. Now, most of these codecs are using generic codec described above. However, some of the packets use fields that are unique to them so i didn't feel necessity to add their encoding and decoding functions into generic codec and decided to implement their codecs by hand.

- Node
  Node part of the plugin is where all the Minecraft logic is implemented. Basically, there is 1 actor created per packet which handles all the processing that needs to be done for given packet. Classes of these actors are then extracted from the plugin and instantiated in the server, thus giving server the control over them.

As it turned out, it was quite simple to implement Minecraft functionality in Specus. I think that for the most part, it was thanks to the usage of Actors. It is simple to reason about a system once we break it down into message passing between entities that don't depend on each other. It is also due to the fact that Minecraft the game is not very complicated, especially considering the goals i chose. But that is ok, as the main purpose of this paper and this project was to get familiar with Actor systems, Scala and learning how to write an extensible server.

7.4 IO & Clients

In order to create a simple to use system, it is important to create right abstractions of IO operations and entities. In Specus, every connected client is represented by session id. `SessionId` is a simple token, that is passed around when message is read or being sent. This token is created when a new client connects and is associated with `Session` object.

`Session` object is abstraction, which knows how to write objects to connection with client and how to close the connection. These objects are stored in `SessionManager` and should never be visible to anyone else. When server needs to write some data into a connection, it should ask `SessionManager` by passing it a `SessionId` and data to be written.

What this means is that any part of the system doesn't need to know anything about how actual the IO is performed. It only needs a `SessionId` and data to be written. `SessionId` tokens are immutable and serializable, so they can be easily passed around.

Description of how actual IO is implemented can be found below.

7.5 Tools and reasoning behind them

7.5.1 Redis

Redis is a high performance key-value database that is used in Specus. Unlike most of key-value databases, it supports a number of different value types:

- string: A binary safe string type, which can be used for storing binary data with efficient random access. In Specus it is used for storing Minecraft map chunks.
- hash: A hash map type, which is optimized for storing multiple key-value pairs. It is used for storing data about clients in Specus.

- set: A typical set data structure, used to store client ID's in specus.
- sorted set: A typical set, except sorted.
- list: a linked list data structure.

It is used to store state, so that it can always be accessed from any node. It uses Scala-redis library, which is unfortunately synchronous. However, thanks to Akka actors it was very easy to wrap the synchronous client into an Actor to create asynchronous interface.

7.5.2 Netty

Netty is high performance library for network IO. It's abstraction over Java io functionality, which supports both TCP and UDP. In Specus, it is configured to use asynchronous processing using non blocking nio functionality. It uses 3 main components:

- specus encoder
- specus decoder
- specus handler

Netty gets these components on start up and uses them transparently when they are needed. User doesn't have to deal with those, he simply writes and read objects from the channel. Both encoder and decoder use `Codec Repository` when looking up codecs for packet encoding and decoding. Codecs are loaded on start up from server plugins.

- Specus Encoder
  Specus encoder is a class that takes an object and using the `getClass` method looks up an appropriate codec for the class. It then uses the codec to encode the object into an array of bytes.
- Specus Decoder
  Specus decoder works similarly to Specus encoder, except it looks up codecs by byte identifier (every packet type in Minecraft protocol is prefixed by id byte). Obviously, we assume that client always sends valid data. If it didn't we wouldn't be able to recovery from it anyway.
- Specus Handler
  Specus handler contains callbacks which are invoked on certain events in the system.

- channelConnected
  This event is invoked when a new client is connected to the server. Netty allows user to set a so called `attachment`, which is available every time an event is invoked on specific channel. I use this opportunity to create a new `Session` and `Session ID` and then store the `Session ID` as an attachment. We also send `ClientConnected` notification to the plugin system, in case some plugin is interested (for example Stats plugin).

  - channelClosed
    This event is invoked when a client connection is closed. We send notification to both plugin system and to node, so that it can clean up after client and then we destroy session associated with the client.
  - writeRequest
    This event is invoked when data are being written into the channel. We just use this callback to sent notification to the plugin system.
  - messageReceived
    This event is invoked when a packet is parsed by Netty. We need to associate it with the client somehow, so that we can respond it. Luckily, we saved `Session ID` as an attachment and we can withdraw it now. We sent the parsed packet and session id to both plugin system (so that it can be registered by stats plugin).

8 Conclusion

In the last part of this thesis, I will try to compare my implementation of server with official implementation and talk about how I tested the project specification. I will try to review and judge decisions I 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.

8.1 Comparison to official server

Unfortunately, official implementation of the Minecraft server is not open sourced and the actual compiled jar is obfuscated, so there is little 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 between nodes or split the map chunks to different servers. Synchronization would add a lot of additional traffic and complexity while splitting chunks would make for a very vulnerable design. 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 as of 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 [8] 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.

8.2 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 platform and architecture, but testing of complete server with Minecraft plugin 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

[8] www.bukkit.org

would have to write my own client in order to test it properly, which would by itself probably take as much time as the entire server implementation.

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 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[9] so testing with multiple client instances would be 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.

8.3 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.

8.4 Review of used tools

8.4.1 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

[9]Build tool for Scala projects.

reason about and easy to get right. In Scala, one also almost never writes any looping code (for example for iterating over collections), which eliminates a whole set of bugs that can one introduce to system. Also, thanks to Scala powerful type system, i had to use type casting only once (in implementation of type codec), and compiler caught a lot of errors during compile time.

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. On the other hand, one has to pay attention to the fact that Java libraries usually are written using immutable objects, so it requires more attention to keep track of all the possible thread-unsafe entities.

8.4.2 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. It has very useful API, which allows user to execute a number of different `Futures` and then invoke different function when all those 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 chunks in a future, then we wait until they are all finished and then we send player the instruction to spawn.

8.4.3 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 and threading that goes along with it.

8.5 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[10] 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.

8.6 What have I learned

[10]domain specific language