design.org

Tomas Herman

2012-02-28 Tue

Contents

1 Design

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.

1.1 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 sent are formatted using logical packets. There is a number of different packet formats. Every packet is prefixed with unsigned byte which indicates the type of packet, which ultimatelly 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.

1.1.1 Data types

For the most part, minecraft packets consists of only few well defined data types. Some packets however use ad-hoc formatted datastructures. 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

1.1.2 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, datatype and value itself) of variable length. Every triplet begins with a byte. Top 3 bits (with `0xE0` mask) of the byte encode the datatype of value while the bottom 5 bits (mask `0x1F`) encode the id of entity. The value itself depends on the datatype 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 triplet
0x110	int, int, int

1.2 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.

1.2.1 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 avalible 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` superclass that all packets need to extend and so on.

1.2.2 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 abotu 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 for given byte is registered (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 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.

1.2.3 Node

All the actual business logic is done in nodes. Nodes are independent JVMs runing 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 avalible nodes and starts sending messages to it.

In nodes, message processing should be done either in stateless fashion, or the state should be persistet in some sort of database as the messages are sent randomly to the nodes. In order to do 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 the 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.

1.2.4 IO handling

1.3 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.

1.3.1 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 kind of messages which indicate what events are happening in the system.

On the other hand, node part of the plugin usually consits 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 entry point class as she wishes. For example, in server subproject 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.

1.3.2 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 either validate that all dependencies are either fulfilled or there is something missing.

Once all the dependencies are checked, entry point clesses 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 pregenerated. 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.

1.3.3 Communication among plugins

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

As mentioned above, in plugin descriptor there is a field specifing 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 avalible, as the plugin system already verified that all plugin dependencies are avalible at this point. The received reference is simple `ActorRef`, as specified in Actor part of this paper, thus alowing user to simply send messages to it.

1.3.4 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 processor 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 datastructure, so when other plugin ask for this data, it can be very efficiently sent (basically it just sents a referrence to the map) to it without worrying about somoneone mutating it and thus desotrying the consistency of data.

1.3.5 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.

1.3.6 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 known packets that Minecraft supports, aswell as their codecs. Now, most of these codecs are using generic codec described in this paper. However, some of the packets use fields that are unique to them so i didn't feel necesity to add their encoding and decoding functions into generic codec and deided to implemenet 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.

1.4 IO

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. Session id 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 IO is performed. It only needs a Session id and data to be written. Session id tokens are immutable and serializable, so they can be easily passed around.

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

1.5 Tools and reasoning behind them

1.5.1 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 datastructure, 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.

1.5.2 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 nonblocking 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 plguins.

1.5.3 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.

1.5.4 Specus Decoder

Specus decoder works similiary to specus encoder, except it looks up codecs by byte identifier (every packet type in Minecraft protocol is prefixed by id byte). Obviously, we asume that client always sends valid data. If it didn't we wouldn't be able to recovery from it anyway.

1.5.5 Specus Handler

Specus handler contains callbacks which are invoked on ceratin 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 avalible every time an event is invoked on specific channel. I use this oportunity 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 sytem 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).