Improving OpenTTD@Large

Adding a massive multiplayer component to an RTS-game

Thesis

July 2nd, 2013

Delft University of Technology / The Hague University of Applied Sciences

Author: Lucas van Dijk

Improving OpenTTD@Large

Adding a massive multiplayer component to an RTS-game

Thesis

July 2nd, 2013

Author:
Lucas van Dijk

Graduation position at:
Delft University of Technology
Faculty of Electrical Engineering, Computer Science and Mathematics
Parallel and Distributed Systems Department
Mentor:
ir. Otto Visser

Educational Institute:
The Hague University of Applied Sciences
Electrical Engineering Department
Mentors:
ir. ing. Harry Broeders
ing. Jesse op den Brouw

Abstract

This is a report about a graduation project performed at Delft University of Technology. To test some of their gaming related research in a real life environment, they are setting up a platform called @Large. This platform will add massive multiplayer components to several games. The first game to support this platform is OpenTTD, an open source real time strategy game.

The goal is to have four different multiplayer types for OpenTTD: a quick game, the normal game, a scenario and the "unlimited game". In a quick game the player needs to achieve certain goal within 15–20 minutes, the normal game is almost the same as the default OpenTTD multiplayer with a few additions, scenario games have scripted events, and the player needs to achieve one or more goals, and an unlimited game is a infinite map with an infinite amount of players with a different set of rules: the number of actions a player can perform on a single is limited, and new technology becomes available when certain achievements are unlocked.

This goal of this graduation project is to bring the @Large platform a bit closer to this goal for OpenTTD. Things to implement in this thesis are the possibility to actually select a type of multiplayer game, and a game independent achievement system for @Large, which can be used as base for the unlocking mechanism in the unlimited game. Early in the project it was also decided to refactor a lot of code to C++.

The project went rather successful. The achievement system does not have an implementation yet, but the designs are available. The reason the achievement system is not finished yet is because the refactor to C++ cost a lot of time.

Preface

Thank you for reading this report about my graduation project! This is the final project of my bachelor Electrical Engineering at The Hague University of Applied Sciences. During my study I found out I mostly liked the computer science parts of Electrical Engineering. This is the reason this project is more a computer science project than Electrical Engineering.

Another reason is because I am starting my master Computer Engineering next September. And Computer Engineering contains courses on high performance and distributed computing, I hoped to learn something that would come back in Computer Engineering and be little bit more prepared when my master starts. Although I do not think I have learnt a lot that I can reuse this September, the discussions about why a certain paper was good and why another is not that good were definitely interesting.

Furthermore, I would like to thank Harry and Jesse for teaching the most awesome courses in Electrical Engineering, and doing a really great job with it. Harry for the personal guidance and the help to make this three year program possible, and Jesse for the awesome stories about the very first computers and microprocessors and the rants about Windows. And of course Otto, for the guidance in this project and the good advice when I needed it.

Readers are expected to have knowledge about software engineering and C++. Hope you will enjoy it!

Lucas van Dijk

Contents

Chapter 1

Introduction

OpenTTD is an open source clone of the game Transport Tycoon Deluxe: a management game where the player has to manage a transport company. In a map with different cities, oil rigs, iron ore mines, oil refineries, steel factories, and more places where resources are gathered and other goods are produced, the player has to earn money by transferring passengers, resources or goods. This can be done by setting up bus lines between cities, building a railroad track between a mine and a factory, or any other transportation method like ships or by plane.

The original game was written in x86 assembly, and this gave a lot of problems with newer operating systems. So, at some point, a community reverse engineered the game, and rewrote it in C and C++, and the result was called OpenTTD. They also added a lot of improvements, and added features like multiplayer over LAN and internet.

1.1 OpenTTD as a research test platform

At the parallel and distributed systems department of Delft University of Technology, they have several gaming related research projects. Think of content and level generation, area of simulation (how to simulate only parts of the map where players are looking) and policies when to start or stop new servers. They want to test this research with a real world game, and they have chosen OpenTTD for this. OpenTTD is a good choice because it is a comprehensive real-time strategy (RTS) game and it is also open source.

The goal is to have four multiplayer game types for OpenTTD:

- short game: achieve a certain goal within 15–20 minutes, or before an opponent does. The type of game is meant to be played when limited time is available, for example in the bus or in a lunch break;
- normal game: a game type which is almost the same as the default OpenTTD multiplayer. One or more players control a company and try to make as much money as possible. There is a maximum of 16 companies in the same game, which can be controlled by a maximum of 255 players. The maximum map size is 2048x2048 tiles, and there is no time limit, although nothing new happens when the year 2050 has been reached in OpenTTD. New technology comes automatically after a fixed period of time;
- scenarios: games with scripted events, where the player must achieve one or more goals to win;
- persistent (unlimited) game: an infinite amount of companies on a single infinite map. In this type of game the notion of time does not exist. Players need to unlock certain achievements to unlock new technology. The number of actions the player can perform per day will also be limited.

In the end, there should be a complete community around this "@Large" platform, with statistics about the played games, achievements for players, replays, information about the companies in the persistent game and a lot more. And probably more open source games will be modified to support this @Large platform in the future.

1.2 The Project

The goal of this graduation project is to implement several things, to set the OpenTTD@Large project a bit closer to the end goal described in the previous section.

Things to implement include:

- Make it possible to actually select a type of multiplayer game. The server should handle these requests for a certain game type, check if there are any servers available for this game type, and send the user a hostname and a port of the destination server;
- build an achievement system for the @Large platform. Think of an efficient way to synchronise the progress in a game with the @Large server. It should support multiple games, and should also allow achievements which are game independent;
- Unlock new technology in OpenTTD based on unlocked achievements, when playing the persistent game.

1.3 Requirements

The jobs listed above will be prioritized according the MoSCoW method [4].

Must Have: The following items must be completed to succesfully finish this project.

- Make sure a player can join a game with a certain game type.
- Achievement system for the @Large platform

Should Have: If there is any time left after the "must have" requirements are finished, the following items should be completed.

- Unlock new technology based on unlocked achievements.

Could Have: If all "should have" requirements are finished, and there is still time left, then these items could be completed.

- Refactor some of the current code to C++.
- Logging system for replays and bug tracking.

Would Have: The items below would be nice to have, but will probably not be done in this project.

- Think of a way that prevents opponents from blocking each other.
- Simple website with user profiles and achievements.

1.4 This document

In the following chapters the new common @Large library will be discussed, the modifications to OpenTTD and the achievement system, and afterwards the conclusion and recommendations.

Chapter 2

Library for @Large

In the early stages in the project, is was decided to redesign the common "@Large library" in C++. The goal of this library is to provide a nice structured, object oriented, high level API for the @Large platform, which should be more future proof than the current C library.

The reason this "could have" requirement was started, is because multiple people had the opinion a refactor to C++ could benefit the @Large platform. Another member of the group also already started refactoring small parts to C++. To prevent double work, the refactor to C++ has been scheduled a bit earlier.

This "@Large library" provides all common functionality needed by games which want support for the @Large platform. This new design enables the programmer to easily add new kinds of messages used to communicate with the @Large server, and it is easily extended with game specific functionality. This also benefits the first part of the whole project: adding new messages and features is easy, while keeping the code maintainable.

2.1 Requirements

Before listing the requirements for a common library, lets think about the use cases. For a general client, there are some common use cases that each @Large game client should support. Players need to authenticate themselves on the @Large server, list the available games, and join one. And after playing a player is able to logout.

Besides the gamers as clients, the @Large server also has game servers as clients. These servers need to advertise themselves to the @Large server, so the @Large server knows which games are currently available. A game server is also able to notify the @Large server when it shuts down. In Figure 2.1 an UML use case diagram is shown with all common use cases.

The requirements for the library are stated below, keeping the use cases in mind.

- The library must handle the connection to the @Large server.
- The library must handle authentication.
- The library must provide a high level API for communicating with the @Large server.
- The library must be easily extended, games probably want to add some extra functionality.

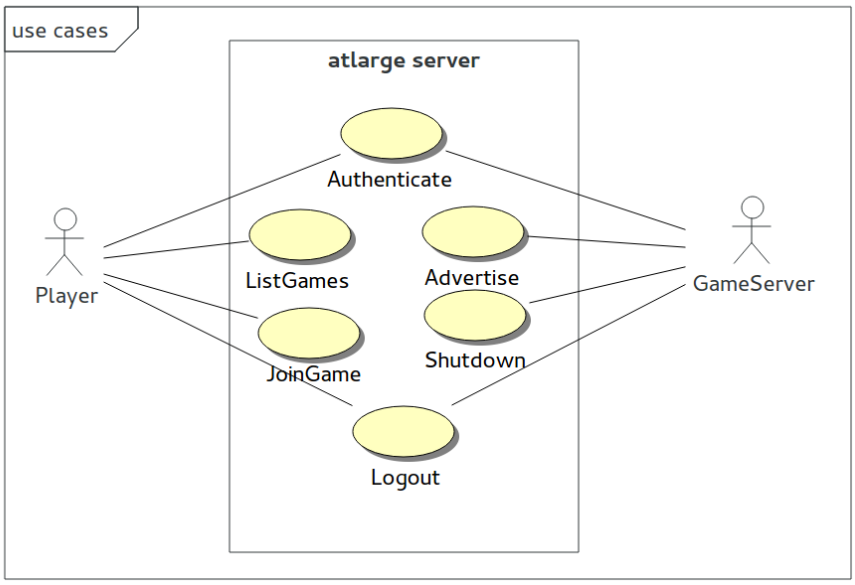

Figure 2.1: UML use case diagram of the common use cases.

2.2 Library design

In this Section is explained how the library is designed and why certain choices were made.

2.2.1 Multiple libraries

The common functionality is split in two libraries: one named *libatlarge-util* and the other just *libatlarge*. The first one is actually not really related to the @Large platform: it contains a few classes to abstract the C socket API, a class which sends and receives data following a specific protocol, which includes some fields for the length of the data and more, and some classes useful for logging. This split was made because other parts of the platform could also benefit from a C++ socket class, and did not require all the @Large specific protocol handling.

The second library is the main @Large library: this library manages the connection to the @Large server and deals with the underlying JSON [5] protocol. For game clients it provides functionality to authenticate users, and list the different game types and configurations. For game servers (which is not the @Large server, but a server hosting a specific game), it provides functionality to advertise themselves to the @Large server.

2.2.2 The library libatlarge-util

This library consists of two main components: the networking part and the logging part.

Networking

The networking part includes a class for abstracting the C socket API. It supports both IPv4 and IPv6, TCP and UDP, and provides some convenience functions like `send_all`. It was created to simplify network programming a bit, because it is no longer needed to deal with the different C structs for network programming.

Besides the `Socket` class, it also provides another class called `ProtocolHandler`. This class can be used to send and receive data formatted according to the following protocol: before the real data, some extra fields have been added. This extra data contains the length of the data, a field if the data is compressed with gzip (and thus needs to be decompressed), and the protocol version. Listing 1 displays the struct with all fields that are prepended to the data.

This protocol is used because the @Large platform will probably deal with a lot of different clients in the future, and an out of date client trying to connect to the @Large platform is probably not going to be a rare event. To avoid crashes on both the client and server side, caused by different versions of the protocol, out of date clients should be notified to upgrade to a newer version. Furthermore, compressing the data with gzip can save bandwidth, and the server or client should know whether the data needs to be decompressed.

```
1 struct PacketHeaderStruct
2 {
3         uint16_t check;
4         uint32_t length;
5         uint8_t version;
6         uint8_t gzipped;
7 };
```

Listing 1: Struct with the fields prepended to the data

The *check* field is used to check if the received data is really the beginning of a new packet, the value should always be $FFFF_H$. *Length* contains the number of bytes this packet contains (excluding these extra fields), *version* is an integer for a protocol version, and *gzipped* is a flag whether the contained data is compressed with gzip or not.

Logging

Logging is used throughout the library and the clients to check the behaviour of the program, and check incoming or outgoing data. Using a standard logging API, it is possible to control the output, for example write it to a file, send it to the standard output or ignore the logging output completely. The logging library is designed to be flexible and extensible, it allows the programmer to easily control what to do with the log entries, and how it should be formatted.

It is possible to use Apache *log4cxx* [3] library, but it was deemed a bit overkill, and would have added an extra dependency to the library. The logging library described below was originally written for an other project [12], and could be reused in this project with few modifications.

The class `Logger` is the main entry point for logging. It provides shortcuts to log with a certain level, and also implements a C++ stream API, which allows the programmer to log messages like printing to `std::cout`. The `Logger` class creates instances of `LogRecord`, and passes these objects to the `LogHandler`. Each `LogHandler` has a certain formatter function, which transforms the `LogRecord` instance to text. Any derived class from `LogHandler` can then decide what to do with the text, either writing it to the standard output or to a file. An UML class diagram is shown in Figure 2.2.

This design is based on the idea of "separation of concerns" [9]. The `Logger` class only provides the API to create `LogRecord` objects, and does not care what any `LogHandler` does with them. This allows users of this library to write their own `LogHandler`, and do whatever they want with the records, without needing to modify the original library.

This design is better than providing a single `Logger` class, which includes both the logging API and the log handling code. This becomes clear in the following example: the user has multiple instances of the `Logger` class, and the log output should go to the same file (not an uncommon use case). With the design where there is no separate `LogHandler` class, both instances have opened the output file, which will probably result in conflicts, when trying to save the file. In the case with separate `Logger` and `LogHandler` classes, this problem does not exists. The user can create one instance of a `FileLogHandler` class, which opens the output file just once, and assign this instance to both `Logger` instances.

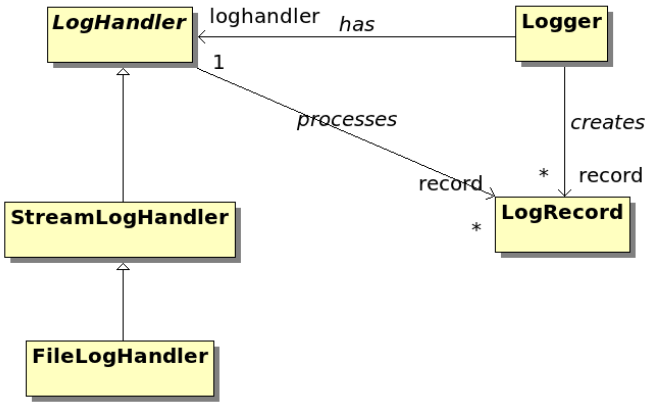

Figure 2.2: UML class diagram with all logging related classes.

2.2.3 The library libatlarge

This library is the starting point for any game which needs to communicate with the @Large server. It manages the connection to the @Large server and uses a JSON [5] library to parse incoming JSON messages, and to create outgoing JSON messages. In this project the Jansson [10] library is used for reading and writing JSON. Jansson is a small C library, and it was chosen because it is also used in other parts of the @Large platform, to keep the dependencies a bit consistent, but also because it is fast and has a nice API. Of course, there are also C++ JSON libraries like JsonCpp [11] and JSON Spirit [13]. JSON Spirit has a dependency on Boost [1], and because Boost is a huge library it was discarded early, and while JsonCpp provides a nice API and has some convenient C++ features, it is mostly syntactic sugar, and does not provide any real benefit above jansson.

This library builds upon the libatlarge-util library, and uses a lot of its components. An UML class diagram with all classes can be seen in Figure 2.3. The `Socket` and `ProtocolHandler` classes from the *libatlarge-util* library are also included in this figure.

Clients and Servers

One of the requirements stated in Section 2.1 is that the library should handle the connection to the @Large server. This why the concept of a general *client* has been introduced, which is represented by the class `Client`. For the @Large server, both game clients as game servers are client. The `Client` class contains functionality for both game clients and game servers.

This class has an instance of the `AtlargeProtocol` class to parse incoming messages and send messages, provides some basic functionality like sending a ping (expecting to receive a pong), and a function which polls the server for new notifications. That last function should be called frequently, for instance in a game loop.

If a game needs to add some extra functionality, the programmer can easily extend from the `Client` class and override or add functionality.

Protocol

The protocol handling is done in the `AtlargeProtocol` class. This is a layer on top of the protocol described in Section 2.2.2, and therefore this class does not need to care about the protocol version and gzipping. The code in this class is not integrated into the `Client` class because the idea of "separation of concerns" also applies here. The `Client` should care about sending and receiving messages in the right order, not how they are sent or received.

The protocol used to communicate with the @Large server is based on JSON [5]. The protocol is built upon two concepts: the envelope, and the message. The reason for this split is to provide a consistent method to determine the kind of message an envelope contains. One of the requirements stated in Section 2.1 was to provide a high level API. Using the factory pattern [2] it is possible to dynamically create the right `Message` object at runtime, based on the type of message inside the envelope. This helps providing an API where the

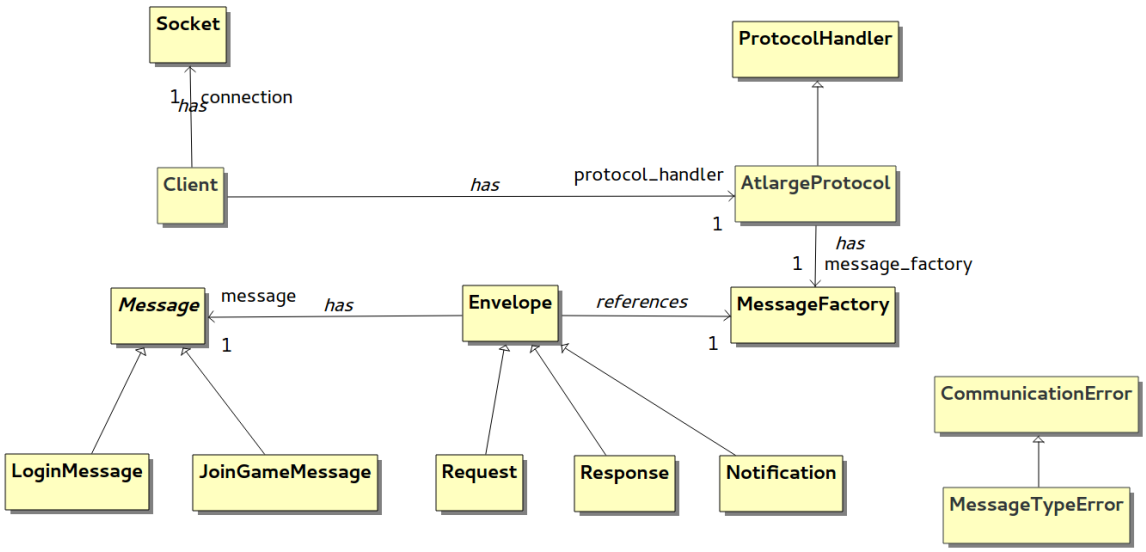

Figure 2.3: UML class diagram of the libatlarge library

programmer only needs to deal with the right `Envelope` and `Message` objects, and not with the underlying protocol.

There are three kinds of envelopes: a request, a response and a notification. Besides the three different envelope kinds, the JSON format of an envelope is always the same. As stated before, the envelope contains the information about what kind of message it contains. The format of a message differs with each message. Each message can use any kind of JSON field it requires. Example JSON structures can be seen in Listing 2 and 3.

```
{"request": {
        "type": "join-game",
        "message": {
                "game": "openttd"
        }
}
}
```

Listing 2: Example JSON structure of a request with a “join game” message.

```
{"response": {
        "type": "connect-to-game",
        "message": {
                "hostname": "openttd-server.com",
                "port": 4242
        }
}
}
```

Listing 3: Example JSON response which instructs the client to connect to a certain server.

The three envelope kinds are represented by the classes `Request`, `Response` or `Notification`, and always contain a `Message` object. An envelope object can be either created programmatically, filling the fields in code, or by parsing a JSON string, which in turn uses the `MessageFactory` class.

If the programmer expects a certain message as response after sending a request, the `AtlargeProtocol` class can also check if the received packet is not an error, and indeed contains the right message.

One thing to note is that it is not the `Envelope` class which contains an instance of the `MessageFactory`, but the

class `AtlargeProtocol` (the `Envelope` class does the JSON parsing). This is done because the `MessageFactory` needs to know which message type corresponds to which `Message` subclass. The factory provides a method to register message types. The `Client` subclasses are responsible for registering the right message types, and by making the factory part of the protocol handler, each subclass of `Client` has easy access to the factory. This also makes sure that each type of @Large client can decide which kind of messages they accept, and makes it easy to add new kinds of messages for games which need extra functionality. This extensibility is also one of the requirements.

2.3 Message generator

As discussed in the previous chapter, each protocol message has its own class. It can become a tedious task to write all those classes for each message, and to speed up the development a bit, a tool called `messagegen.py` has been written. Python has been chosen because it has a lot of functionality built in, like advanced string formatting, regular expressions, configuration file parsers and more.

2.3.1 The configuration file

This Python script reads a configuration file, and generates the required C++ header and source code files for each message class based on the contents of the configuration file.

The syntax of the configuration file is as follows:

```
[unique-message-type]
description = Description of the message
namespace = The C++ namespace which will contain this message
classname = C++ class name
fields = JSON fields, syntax described below
client_only = Optional, whether the server needs a function handler for this message, default False
```

The syntax is just the basic syntax of an ini configuration file. The *description* field is used as documentation, and the value will be used as a documentation comment in de C++ source files. Because the message classes are part of the @Large library or any game specific @Large library, it is a good idea to put them inside a namespace. This can be configured with the *namespace* field. By default, the parser assumes that this message also needs a function handler when the message is received by the @Large server. If this is not required, it is possible to set *client_only* to True.

The *fields* field requires some extra attention. The format of the message can be defined with this field. It is a comma separated list of type–variable name pairs. Supported types:

- `boolean`
- `string`
- `integer` (32 bit integer)
- `unsigned integer` (32 bit integer)
- `long` (64 bit integer)
- `unsigned long` (64 bit unsigned integer)
- `object<one of the above types>`

Most of the types are self-explanatory, save for the `object` type. This type represents a JSON object with some limitations: the type of the keys is always a string, and all values of the object should be of the same type. These limitations exist because the object is implemented as a C++ `std::map` in the generated C++ code.

It is also possible to define arrays, this can be done by appending `[]` to the variable name. This works for all types except for an `object`. Some examples are shown in Listing 4.

```
[get-game-list]
description = This message retrieves the available games on the @Large platform
namespace = atlarge
classname = GetGameListMessage
fields =

[game-list-response]
description = This message contains the available games on the @Large platform
namespace = atlarge
classname = GameListMessage
fields = string games[]
client_only = True

[join-game]
description = This message sends a request to the server to join a game
namespace = atlarge
classname = JoinGameMessage
fields = string game, object<string> options
```

Listing 4: A few examples of message definitions for the message generator.

2.3.2 Functionality for the server

The current @Large server is written in C, and does not use the *libatlarge* library and therefore does not use the message classes. Porting the server to C++ would be too much out of scope for this thesis. To handle incoming messages, the server uses *gperf* [8] to generate a perfect hash table to map a message type to a unique message code. This message code can be used in a switch statement, where in turn the right function is called which should handle the incoming message.

The configuration file for gperf and the C switch code is automatically generated based on another configuration file with message definitions. Because this original configuration file did not contain enough information for the message generator, and it was not based on a standard configuration format, the new ini based configuration file has been created. Parsing an ini file with Python is a lot easier than writing a custom parser.

But, this is a bit inconvenient because there are now two places where messages are defined: a configuration file for the server and a configuration file for the library. To make the original configuration file obsolete, the message generator script also has an option to generate the gperf configuration file and the C switch code. Only messages with *client_only* unset or set as False will get an entry in the configuration file and the C switch code.

In the future is probably a good idea to change this *client_only* option to an option called *handler* with possible values *client*, *server* or *both*. This would make it possible to generate switch code for game client too.

2.4 Result

The result is a library which should be easy to use, extensible and can be used with both game servers and clients. This library will help to customize games to support the @Large platform. Although it did cost some extra time to write this library, it will probably help the platform in the future with an extensible code base, where other games can easily build upon.

This will also help the rest of the assignments of this project, because the functionality is well structured and can be easily altered. Plus I think the quality of the resulting code will improve by using this library, instead of quickly hacking the desired functionality in the game.

Chapter 3

OpenTTD modifications

This new library should be put to good use, and therefore OpenTTD has been ported to the new @Large library. This includes the creation of a small OpenTTD specific library based on the common @Large library. Also, as stated in Section 1.1, one of the goals is to have multiple OpenTTD game types for the @Large platform. This requires modifications to both OpenTTD and the @Large server, and these will be discussed in this chapter.

3.1 @Large library for OpenTTD

Another custom feature the @Large platform provides for OpenTTD is trading of resources, goods and passengers between players on different servers. More about the trading system will be explained below. This functionality needed the be ported to the C++ library.

To provide these functions, another library has been created. This library contains the following components:

- an `OpenTTDClient` class based on the base `Client` class in the common @Large library;
- new messages to communicate with the @Large server, for example to set up and manage trades, achievement data and more;
- required classes for the achievements (more on that later).

3.1.1 Trading system for OpenTTD

To provide a little bit extra on top of the default multiplayer options of OpenTTD, the trading system between servers was invented. Trading is only enabled in the normal game. This system was designed and written before this thesis, and will not be covered in great detail in this report.

The trading system works, but is not in its final form yet. Currently there is a limit of nine "normal game" servers. Each index 0 to 9 has a hardcoded fixed position in a 3x3 grid, with wrap around. This makes sure the @Large server knows which OpenTTD server is a neighbour of whom. Before a player can trade, he needs to build a trading center on the edge of the map.

To setup a trade, a player can open up the new "Trades" window in OpenTTD, click "new trade", select to which direction (only directions with a trading center at the edge are possible), select what to trade (goods, passengers, resources), and offer a price for it. When the player clicks OK, the client sends a "propose deal" message to the @Large server, which makes sure the proposal will be delivered to the right server. If one or more players accepts the deal, the @Large server notifies both servers that an agreement has been made.

3.1.2 Refactoring

A lot of the code in OpenTTD and the @Large server had to be refactored to use the new @Large library and the new message protocol. This included the basic @Large actions like logging in, the whole trading system, and more. Below are a few examples of refactored functions.

Incoming deal function

This function is called when a *new deal message* is received. This message is sent by the @Large server when both parties agree to a trade. The new version can be seen in Listing 5, the old version in Listing 6. This function creates a new `Trade` object (not shown), and stores it in a global structure. The GUI will update itself and will include this new trade agreement in the list with all trades.

```
void atlarge_incoming_deal(openttd::NewDealMessage* message) {
        logging::stdlog << logging::setlevel<Logger::LOG_INFO> << "Received trade from "
                << message->getSender() << " to " << message->getReceiver() << logging::endl;

        createTrade(
                message->getTradeId(),
                message->getCargoId(),
                message->getSender(),
                message->getReceiver(),
                message->getMoney()
        );
}
```

Listing 5: Refactored version of the a function which is called after both parties agreed to a trade.

```
void large_incoming_deal(json_t* message) {
        json_t* cargo_json;
        json_t* tradeID_json;
        json_t* money_json;
        json_t* receiver_json;
        json_t* sender_json;
        AtLarge_PlayerID receiver;
        AtLarge_PlayerID sender;
        const char* key;
        json_t* value;
        uint8_t i;

        cargo_json = json_object_get(message, "cargoID");
        money_json = json_object_get(message, "money");
        tradeID_json = json_object_get(message, "tradeID");
        receiver_json = json_object_get(message, "receiver");
        sender_json = json_object_get(message, "sender");

        receiver = (AtLarge_PlayerID)json_integer_value(receiver_json);
        sender = (AtLarge_PlayerID)json_integer_value(sender_json);
        printf("Received trade from: %lu to: %lu\n", sender, receiver);

        createTrade(json_integer_value(tradeID_json), json_integer_value(cargo_json), sender, receiv
                json_integer_value(money_json));
}
```

Listing 6: Old version of a function which is called after both parties agreed to a trade.

Accepting a proposed deal

The client library contains a function which sends the *accept deal message* to the @Large server. This message is sent when a player clicks the "accept" button in OpenTTD. The new library requires a lot less code because a lot is abstracted away. The new version can be seen in Listing 7 and the old one in Listing 8.

```
void OpenTTDClient::respondToDeal(unsigned long long trade_id, bool accepted)
{
        Request message(new DealResponseMessage(trade_id, accepted));
        this->protocol.sendMessage(message);

        this->protocol.expectResponse(SuccessMessage::TYPE);
}
```

Listing 7: The new version of the accept (respond to) deal message.

```
bool atLarge_ottd_acceptDeal(unsigned tradeID, bool accepted) {
        bool res = false;
        char* acceptMsg;
        char* answer;
        size_t msgLen = strlen(ACCEPT_MSG) + 30;

        debug(1, "Accepting deal %u: %u", tradeID, accepted);

        if (!checkConnection())
                return false;

        acceptMsg = malloc(msgLen);
        if (acceptMsg == NULL)
                fatal("Out of memory");

        snprintf(acceptMsg, msgLen, ACCEPT_MSG, tradeID, accepted);

        writeHeader(strlen(acceptMsg));
        bytesWritten = write(sockfd, acceptMsg, strlen(acceptMsg));
        if (bytesWritten < 0) {
                perror("ERROR writing to socket");
                if (close(sockfd) == -1)
                        perror("Couldn't close socket");
                sockfd = 0;
                free(acceptMsg);
                return false;
        }

        free(acceptMsg);
        answer = readMessage(sockfd);
        if (answer == NULL)
                return false;

        if (answer[0] == '{') // TODO
                res =  true;

        free(answer);
        return res;
}
```

Listing 8: Old version of the accept deal function in the client library.

3.2 Multiple game types

In this section will be discussed how the graphical interface to select the game type and the required server side functionality has been designed and implemented.

3.2.1 Requirements

First of all, a few requirements for this system has been stated:

- the player must be able to select the game type;
- when the player starts a game, he must join the right type of server;
- OpenTTD servers should notify the @Large server what game type they are hosting;
- the @Large server should remember which game servers host which type.

3.2.2 Implementation

Game servers

It all starts with a game server hosting a certain type of game. At the start up of these dedicated OpenTTD servers, they advertise themselves to the @Large server, using the *advertise* message. With this message they include the gamename (in this case "OpenTTD@Large") and a game type, which in the case of OpenTTD@Large is one of "quick-game", "normal-game", "scenario" or "unlimited-game".

An OpenTTD game server can be started with the following command:

```
./openttdAtLargeServer -x -c server_conf.cfg -D [::]:4000 -u user -p password
```

With:

- `-x` - Do not overwrite the configuration file;
- `-c server_conf.cfg` - Specify the configuration file;
- `-D [::]:4000` - Listen on IPv6 address (IPv4 also possible), port 4000;
- `-u user` - @Large username for the server;
- `-p password` - @Large password.

Each game server type has its own configuration file, and in the configuration file is also the game server type specified. Additionally, it is possible to enable custom OpenTTD add-ons for this game server, for instance the different scenarios.

@Large server

When the @Large server receives an "advertise" message, it checks which game is specified. Currently, OpenTTD is the only supported game, so currently a hardcoded check is performed to see if the game is "OpenTTD@Large". If it is not "OpenTTD@Large" an error will be returned. If it is "OpenTTD@Large", an OpenTTD specific function will be called, which should handle the rest of the advertise message.

To store all OpenTTD@Large game servers, multiple hash tables are used: for each game type a separate one. Each hash table is also an element of the "main" hash table, with its game type as key. This allows to easily retrieve the right hash table to store game servers in for a given game type.

Because the @Large server is written in C, and C does not contain a lot of data structures by default, we use Glib [7] to provide our hash table data structure. Glib was already a dependency of the server, so it was a natural choice to reuse this library for this part of the server too.

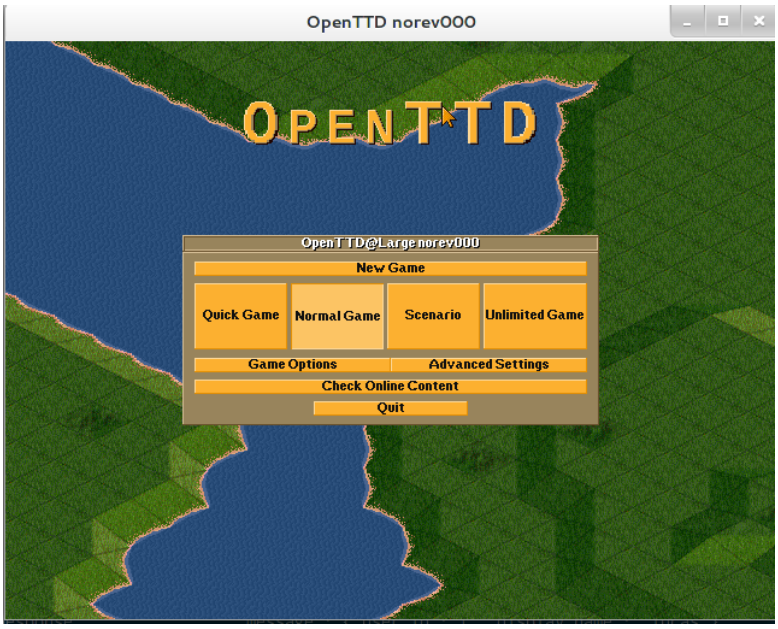

Figure 3.1: OpenTTD@Large intro window.

The choice to store the servers in hash tables has several reasons: first of all, it needs to be backward compatible. This means that for a normal game, the trading system should still work. Because the location of the server in the 3x3 grid is hardcoded and based on the server number, it is required to use a data structure where the key to retrieve the element remains the same.

For example, it is not possible to use a `GArray` [6] (a dynamic array implementation) as data structure, because when an element is removed, it moves the last element to the position of the removed element. This could mean for a normal game that one server is now located on the other side of the 3x3 grid, which makes all trades invalid.

Furthermore, hash tables have the best time complexities for lookup, insertion and deletion compared to linked lists, binary search trees and dynamic arrays. The lookup, insertion and deletion time complexities of a hash table are all $O(1)$ on average and $O(n)$ in the worst case.

OpenTTD

The player should of course also be able to choose which game type he wants to play. OpenTTD has a very basic window and widget system, and the main introduction GUI has been altered to offer the game type choice. This can be seen in Figure 3.1. When an user clicks “new game” a “join-game” message will be sent to the @Large server with the requested game type, and then the @Large server will return a hostname and a port, the address of the OpenTTD game server.

3.2.3 Result

The result is working system for multiple types of game servers. It can be extended with other game types, by just adding an element to the “main” hash table. On the client side the player can easily choose which game type he wants. It does not look very attractive yet, because the parallel and distributed systems department lacks graphic designers. Improving the graphics is something for the future.

Chapter 4

Achievement system

One of the requirements of this thesis was to create an achievement system for the @Large system. This will also serve as base for unlocking new technology in OpenTTD. The design of this system will be discussed in this chapter.

4.1 Achievement system for all games

This achievement system should be game independent. Of course it should be possible to have achievements which can only be unlocked in one single game, but the API should be standardized across the @Large platform. In this section the game independent API will be discussed, which is part of the common @Large library.

4.1.1 Requirements

Like always, it is good to first think about the requirements of this system:

- achievements can either be game independent or linked to a game;
- the unlockable achievements should be stored in a database on the @Large server;
- user progress should be stored in a database on the @Large server;
- there should be an easy API to check in-game if an achievement is unlocked;
- synchronising the achievement data should not be too heavy on the @Large server.

4.1.2 Database Design

It is good practice to prevent double data in your database. This idea has been kept in mind with the achievement database design. The design looks like this:

- an users table, contains for example the credentials to login, and more user data;
- a games table, contains the name and description of all playable games on the @Large platform. Currently only OpenTTD;
- an achievement table, contains all unlockable achievements, with a name, description and more;
- an achievement progress table. Contains the progress of a certain user for a certain achievement.

The records are linked together with foreign keys. This is visualized in Figure 4.1.

This design prevents double data. For example, without an separate progress table, and instead the progress would be stored in the achievement table. a separate record for each achievement–user combination would be inserted in the table, which results in a lot of double data.

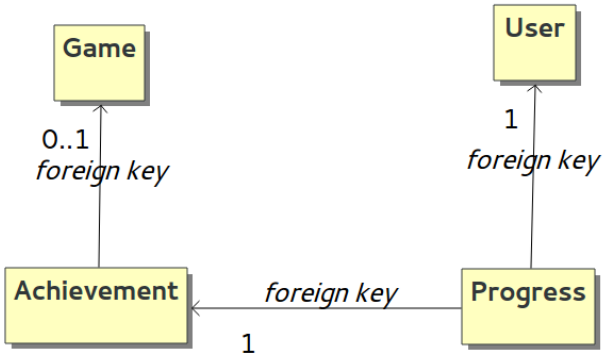

Figure 4.1: The database design for the achievement system.

4.1.3 API Design

The API design has got many iterations, but the current form is described below, and is visualized in Figure 4.2.

The design is similar to the system for the messages. There is a class `AchievementFactory`, which can instantiate subclasses of the base `Achievement` class, based on an *achievement ID*. This ID is the same ID as the record in the database. Game specific libraries should register their achievement classes, to make sure the factory knows which ids maps to which subclass of `Achievement`.

To retrieve the achievement data for a certain game from the @Large server, a program needs to follow the following procedure:

1. Send a "get-achievement" message;
2. the @Large server sends a response with the achievement data. It contains the achievement ids, names, and descriptions;
3. instantiate the achievement objects with the `AchievementFactory`, based on the received ids;
4. store the received name and description in the corresponding `Achievement` instance. Hardcoding the achievement name and description in code is probably not a good idea because that would limit the flexibility a lot.

This procedure will be handled by the class `AchievementManager`. This class shall store the created `Achievement` instances in a C++ `std::map` with the achievement ids as key for easy retrieval. This also makes sure there is a central location where all achievements are stored.

`Achievement` classes have the following properties:

- they contain the achievement name, description and id;
- they have a method to check if a certain user has unlocked this achievement and a method to get the user progress.

`Achievement` subclasses should probably proxy the call to get the user progress to a function which has access to the game internals. With a proxy to the game internals the @Large server has no part anymore in retrieving the achievement status. Of course, the progress needs to be synchronised at some point, but this has not been given a lot of thought yet, and it is definitely something to think about in the future.

This design has several advantages: it is very flexible, games can register their own achievement classes, and are not restricted on how they check if a user has reached the goal for an achievement. All components are loosely coupled which makes it easy to test. And all the achievement data is easily accessed within games, without keeping the @Large server too busy.

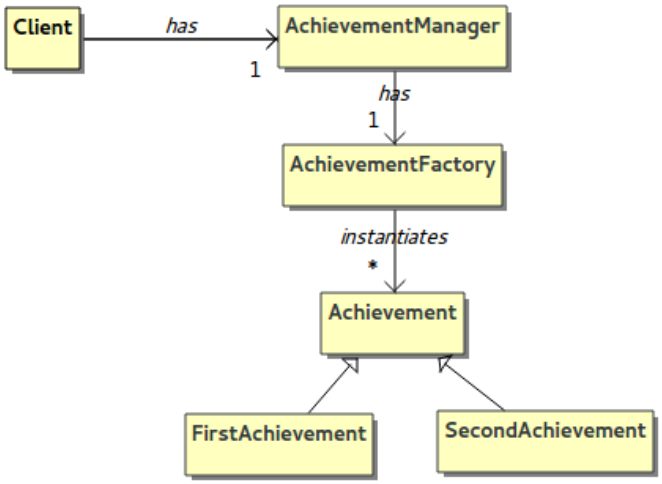

Figure 4.2: UML class diagram of the common achievement API.

4.1.4 Implementation

There is a simple test implementation for achievements, but it is based on an old design, and due to time constraints it was not possible to update the code.

4.2 Unlocking new technology based on achievements

With the achievement system in place, it is possible to think about how to unlock new technology in OpenTTD based on these achievements. Due to time constrains this system also does not have an implementation yet, but the basic idea is as follows, visualized in Figure 4.3:

- There is an `Unlocker` class. This class depends on certain achievements, and when all achievements are unlocked, it unlocks new technology (for example, all vehicles with a speed below 50 km/h).
- A class `UnlockerPool`. This class stores all `Unlocker` instances.

The dependencies of an `Unlocker` instance are specified using the achievement ids. The `Achievement` instances can be obtained with the `AchievementManager` class. This makes it possible to check if a certain achievement is unlocked.

In OpenTTD, there will be one single `UnlockerPool`, which will contain all `Unlocker` instances. This makes iterating over the available `Unlocker` instances easy. All `Unlocker` instances should frequently check if the achievements they depend on have changed their status, which could result in new technology for a certain user.

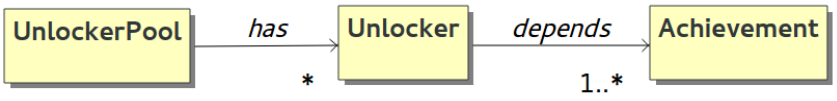

Figure 4.3: API design for achievements.

Chapter 5

Conclusions and Recommendations

5.1 Conclusion

After writing a lot of lines of code, debugging sessions and testing, this graduation project has come to an end. The "must have" requirement to support multiple game types in OpenTTD has been met. There is now support for multiple types of servers, and players can select the game type in the intro screen.

The "must have" requirement to create an achievement system for the @Large platform has been partly met: there is a great foundation for an achievement system and the database design is ready. But the implementation is currently still based on an old design, and needs refactoring. Besides, the OpenTTD specific parts are not ready yet, but there is a design on how to unlock new technology based on the achievements.

But, there is a whole new common @Large library, with features like network and protocol abstraction, logging, JSON parsing, error handling, and more. With the new message generator it is easy to extend the protocol with additional messages. This new library should provide a nice structured and stable base for any game or application which needs @Large support. This compensates a bit for the not completely finished must have requirements.

5.2 Recommendations and future work

The @Large platform is far from finished yet, and here are some recommendations for projects in the future:

- Finish the achievement system. The ideas are there, it only needs an implementation. Also think on a way to synchronise the in-game achievement progress with the @Large server.
- Refactor the server to C++. C++ has much more capabilities on memory management, it is often much clearer which part of the program cleans up what object. Besides, the server could also use the *libatlarge* library with all its message classes and other features. No need to hardcode the JSON messages anymore.
- Finish the system to unlock new technology based on the achievements. This brings the "unlimited game" a bit closer.

Bibliography

[1] Boost C++ libraries. http://boost.org.

[2] The factory pattern. http://www.oodesign.com/factory-pattern.html.

[3] Apache Foundation. log4cxx library. http://logging.apache.org/log4cxx/.

[4] Coley Consulting. MoSCoW method. http://www.coleyconsulting.co.uk/moscow.htm.

[5] Douglas Crockford. The application/json media type for javascript object notation (json). RFC 4627, IETF, June 2006.

[6] GNOME Project. GArray Reference Documentation. https://developer.gnome.org/glib/stable/glib-Arrays.html.

[7] GNOME Project. Glib Reference Manual. https://developer.gnome.org/glib/.

[8] GNU. Gperf – perfect hash table generator. https://www.gnu.org/software/gperf/.

[9] Walter L. Hürsch and Cristina Videira Lopes. Separation of concerns. Technical report, 1995.

[10] Petri Lehtinen. Jansson JSON library. http://digip.org/jansson.

[11] Baptiste Lepilleur. JsonCpp Library. http://jsoncpp.sourceforge.net/.

[12] Lucas van Dijk. Antsmanager bot. http://bitbucket.org/sh4wn/antsmanager.

[13] John Wilkinson. JSON Spirit C++ library. http://www.codeproject.com/Articles/20027/JSON-Spirit-A-C-JSON-Parser-Generator-Implemented.