Alexander Pagonis, BSc

Bot-Based Continuous Integration for MOBA Games

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Slany

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Slany

Graz, Oktober 2017

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date	Signature

Abstract

This thesis is about enhancing the continuous integration process of a local game development company, utilizing bots. The initial situation is a continuous integration process, where a majority of problems is detected by manual exploratory testing. Throughout the course of this thesis a system is developed that can execute an arbitrary number of configurable tests, each involving computer-controlled players with adjustable behavior, that play the game fully autonomously. The main focus of this thesis is the generation of a system that allows the execution of several tests on demand and populates a database with information about the progression of the tests. Using this system, the programmers of the company receive emails including a feedback about the functionality of their product on a daily basis and can debug more effectively, utilizing a graphical user interface to the afore mentioned database.

Acknowledgements

First of all I would like to thank the company Bongfish who enabled this thesis and provided me with the time to realize it during working hours. Special thanks goes to Stefan Habenschuss, who was always there to help me in case I needed advice and Thomas Richter-Trummer who had an organizational key role.

I would also like to express my gratitude to my supervisor Wolfgang Slany for his support during the creation of this thesis.

Last but not least I would like to thank my family who has always been supportive throughout the course of my studies and Magdalena who has been my bastion of calm for several years now.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

There is a big variety of systems aiming to simplify the realization of big software projects. One of the problems with large-scale software projects is the high complexity of the steadily growing product. The more complex a software gets, the harder it is to keep an overview over the whole system. Therefore it gets harder to extend the software by new features as not only the new feature may be faulty, but it may also introduce faults to other sub-systems. According to Humble and Farley in [16], for large scale software projects it is therefore a necessary step to invest some time into the deployment of automatized systems that take on highly repetitive and easy to automatize procedures. They further state that repetitive tasks are best processed by machines as humans tend to be more error prone when executing the same task over and over. Additionally, when these tasks are executed close to a deadline, the risk of making mistakes is even further increased for humans. Version control systems like SVN[1] or continuous integration systems, such as TeamCity[2] can simplify some aspects of the software development process. However there is no all-in-one solution that can guarantee the viability of any kind of software with just the press of a button.

This thesis describes the generation of a fully automatic test system for computer games incorporating bots. The aim is to generate a system that can automatically test the up-to-date state of the software, as if humans were playing the game. The idea is to execute the system at night, making computer-controlled players play the game fully automatically. While the game is played the generated log-data shall be forwarded to a database. This database shall be accessible using a graphical user interface. In addition, the test system shall provide the programmers with an email, including an overview over the encountered events during the test.

[1] `https://subversion.apache.org/`
[2] `https://www.jetbrains.com/teamcity/`

1. Introduction

This system aims to enhance the continuous integration process of a company creating computer games. The use of this system shall support the quality insurance staff of the company, by automatically testing the game at a daily basis and delivering feedback at the push of a button. In addition, the debugging process for the programmers shall be improved by deploying a system for log-analyzing.

The thesis is structured to initially provide a brief introduction into continuous integration and software testing in Chapters 2 and 3. Subsequently the necessary knowledge about bots and machine learning, to understand the methods used in the project part of the thesis, is covered in Chapters 4 and 5. The remaining thesis covers the implementation of the a previously mentioned system in Chapter 6.

2. Continuous Delivery / Continuous Integration

In the following chapter the terms "continuous integration" and "continuous delivery" are briefly explained, including methods and benefits.

2.1. Definitions

In [22] Pathania defines continuous delivery as a practice in software engineering where production-ready features are produced in a continuous manner.

The term "continuous integration" was introduced by Beck in [6]. Linger et al. define continuous integration in [3] as a process in software development where changes are added frequently, typically on a daily basis, while keeping the software in a working state. In [11] Fowler additionally states that every integration in the continuous integration process is to be followed by a built and testing process by an automatic system in order to detect errors as soon as possible.

In [16] Humble and Farley state that continuous integration is an essential part of continuous delivery. Pathania further states in [22] that the combination of continuous integration, deployments and automated tests can be referred to as "continuous delivery".

In [23] Spencer and Johnston define software deployment as follows:

> Software deployment is the process of installing application programs on a system (workstation or server) and making that application and the data it supports available to the users.

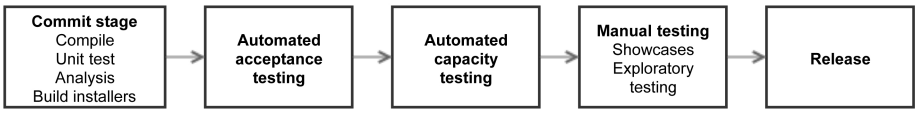


Figure 2.1.: This is an example for a deployment pipeline taken from [16].

Summing up, continuous integration and delivery can be described as paradigms that have emerged from ever-growing software projects due to the necessity of a reliable and more predictable development process.

2.2. Methods

In [16] Humble and Farley introduce the concept of the so-called "deployment pipeline". This pipeline represents an automated tool that manages the build, deploy, test and release process. An example for such a pipeline is depicted in Figure 2.1. Details regarding different kinds of tests are covered in Chapter 3.

Furthermore Humble and Farley state following release anti-patterns in [16]:

- Deploying Software manually;
- Deploying to a production-like environment only after the development is complete;
- Manual configuration management of production environments.

Instead the goal should be:

- A fully automated deployment process that works by simply pressing a button;
- To integrate the testing, deployment and release activities into the development process;
- To automatize the configuration of the system and automatically apply changes from version control.

Humble and Farley also state that every change should trigger an automatic feedback process. The feedback shall be available as soon as possible and the staff should react immediately in case of newly introduced errors.

2.3. Benefits

In [11] Fowler summarizes the benefits of continuous integration as follows:

> Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.

Regarding the benefits of continuous delivery, Humble and Farley state in [16]:

> The principal benefit of the approach [...] is that it creates a release process that is repeatable, reliable, and predictable, which runs in turn generates large reductions in cycle time, and hence gets features and bugfixes to users fast.

Humble and Farley further state in [16] that without continuous integration, the software is likely to be broken until testing or integrating is executed. With continuous integration, changes are integrated and tested constantly. Therefore, bugs are reported immediately making bug-fixing easier and keeping the software in a working state.

2.4. Usage

The project of this thesis resulted from an analysis of the continuous integration process of a game development enterprise. The company is generating content for a computer game that has been in development for several years and has already been published and received several updates. Continuous delivery has been deployed for this project and is applied successfully. Therefore, content is being added continuously and automatic tests are generated that provide immediate feedback upon changes to the repository, using a continuous integration server. However, due to the high complexity of the system, including distributed computing mechanics, high-level game mechanics and other properties of similar complexity, a significant amount of problems is detected in the exploratory testing phase of Figure 2.1. The different kinds of testing, including exploratory tests, are explained in more detail in Chapter 3.2.1. Exploratory testing is a

manual testing phase as the block in Figure 2.1 suggests. Therefore it requires a lot of human interaction. In this project, the exploratory testing is also referred to as "playtesting". It basically consists of humans and sometimes also incorporates bots. General information about bots can be found in Chapter 4. During a playtest, humans generate a session for a game that other humans or bots can join. In this context, bots are only used to compensate for the little amount of available human players. However the bot-system supplied by the company is capable of playing the game fully autonomously. Therefore it seems like an obvious step to try and automatize the exploratory testing phase, utilizing the already available bot system. This is the main goal of the project part of this thesis.

3. Software Testing

The following chapter shall provide an overview over the concept of software testing. This overview includes common Methods, benefits and important details regarding the project part of this thesis.

3.1. Definition

Software testing can be separated into static and dynamic software testing. Hass defines dynamic testing in [14] as follows:

> Dynamic testing is testing of software where the object under testing, the code, is being executed on a computer. [...] Dynamic testing finds failures, that is, situations where the object under testing does not behave as expected.

The author also defines static testing in [14]:

> When we perform static testing we look directly at the written or drawn work products. This means that we are looking for defects directly - those that the producer happened to place there as the result of a human mistake.

Craig and Jaskiel define the more specific term "preventive testing" where test-cases for code are created before the code itself is created in [9] as follows:

> Testing is a concurrent lifecycle process of engineering, using and maintaining testware in order to measure and improve the quality of the software being tested.

This approach is also known as "test driven development" according to [13].

3.2. Methods

Software testing can be separated into white-box and black-box testing. Maidasani describes in [18] that the logic behind a program or unit is known in white-box testing and the aim is to test all paths of the code and maximize the coverage, while comparing the output to a desired state. In contrast to white-box testing the author further describes in [18] that black-box testing focuses on the compliance of the program regarding the specifications and not taking the implementation into account. According to these definitions, white-box tests focus on the functionality of the code while black-box testing is focused on validating the functionality of the product.

3.2.1. Different Kinds of Tests

There is a big variety of software tests. Covering every single test kind would go beyond the scope of this thesis. Therefore following section shall briefly discuss the different kinds of tests which are relevant to this thesis.

Unit Tests

According to Hamill in [13], a unit test is designed to test the behavior of a particular code section. These tests should be performed in an isolated environment to only test the target functionality. The test can be regarded as successful in case the output corresponds to a specified value.

Acceptance Tests

In [12], Goldsmith states that the aim of acceptance testing is to demonstrate that the developed system meets the business requirements. These tests are designed based on so-called "acceptance criteria". These criteria are defined with respect to the following areas:

- What *function* the system must perform;
- *How much* must be shown to give confidence it works;
- *How well*, how system quality will be assessed;
- *By whom*, who should perform tests for confidence we can use it;
- *In what manner*, testing format for confidence the system works.

Capacity Tests

Humble and Farley define the term "capacity testing" in [16] as a test that measures the maximum throughput a system can sustain while maintaining an acceptable response time for individual requests. Throughput, in this context, is the number of transaction of a system in a given time-span.

Integration Tests

Integration tests, as defined by Keller and Plödereder in [17], can be used to test the complete software against the system requirements, running on the target hardware.

Usability Tests

According to Graham et al. in [2] usability tests determine how well a product is understood or how easy to learn or how attractive the software is to the user.

Load Tests

Graham et al. define the term "load testing" in [2] as a test type that measures the behavior of a component or system with increasing load. In this context, load can be an increasing number of users, transactions and similar things, depending on the product.

Exploratory Tests

According to Whittaker in [26], exploratory testing is a form of manual testing. When executing exploratory tests, the user can test the functionality of the application by utilizing any desired feature of the applications. There is no script on what or how to exactly test.

3.3. Testing Model

In [19] Marick introduced the so-called testing quadrant diagram, as exemplified in Figure 3.1. This model provides an overview over the whole testing process and puts the different kinds of tests into context.

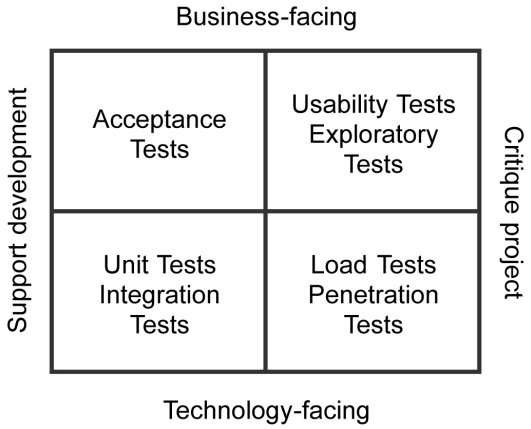


Figure 3.1.: This is the agile testing quadrant diagram by Marick, taken from [21].

According to [19] tests can be separated into business facing and technology facing tests. The separation can be determined by the way the test is described. If the test uses terms from business domain it is business facing. Same applies to technology facing tests. Tests that support programmers during the development process are located at the left half of the diagram in Figure 3.1. Tests that focus on the finished product with the intend to discover inadequacies are located at the right half of the diagram.

3.4. Benefit

Software testing is a necessary step in software development. If no test is executed the developers cannot ensure that the product is in a functional state. Furthermore, Hayes states in [15] that automating software tests speeds up the testing process, which is important in order to be on schedule and it can also save budget that would be used for testing otherwise.

3.5. Usage

All kinds of tests mentioned in Section 3.2.1 are already used by the company to test the software product. Furthermore, a test-driver development process is used. The project part of this thesis utilizes the concept of dynamic black-box testing by trying to automatize the manual exploratory testing phase.

4. Bots

This chapter shall briefly introduce the concept of so-called "bots" in general, as well as in computer game context.

4.1. Definition

It is hard to find a general definition for the term "bot". Definitions for specific kinds of bots exist, such as the Chatbot, defined by Boonthum-Denecke et al. in [1]:

> Conversational System – Chatbot (or ChaterBot or Chat Bot) is a type of computer program designed to simulate an intelligent conversation with a human user.

In the context of a Botnets, Dunham et al. state in [10]:

> Bots are highly adaptable worker bees that do their master's bidding over a broad "net" - in case of bots scattered throughout the global Internet

They further state that the term "bot" originates from the word "robot" which is similar to the Russian and Czech word for "work".

In the gaming industry non-human players are referred to as bots. In this context bots are programs that are designed to play a game. There is a wide variety of approaches on how to achieve this goal. In general the approach on how to best implement a bot is highly dependent on the use-case. Recent research focuses on the use of machine learning in order to teach computers to succeed in different kinds of computer games. The paper [25], released in 2013

by Mnih et. al introduces a method that enables a computer to learn to play a variety of Atari games, based on the graphical feedback of the game.

4.2. Usage

In the project of this thesis, bots are used to play a game all by themselves. They manage sessions, play the game and shutdown the game autonomously. The basic bot system in this project is already provided and is therefore not implemented as part of the project. It consists of several components, which are further described in Section 6.3.1. The aim of this project regarding bots is to enhance their in-game behavior. Their behavior is scripted and extended using basic machine learning approaches as described in Sections 6.3.4 and 6.3.5.

5. Machine Learning

This chapter shall provide a brief overview over the basic principles of machine learning and the methods used in the project.

5.1. Definitions

Mitchel defines the term "machine learning" in [20] as follows:

> Machine Learning is a natural outgrowth of the intersection of Computer Science and Statistics. [...] Whereas Computer Science has focused primarily on how to manually program computers, Machine Learning focuses on the question of how to get computers to program themselves (from experience plus some initial structure). Whereas Statistics has focused primarily on what conclusions can be inferred from data, Machine Learning incorporates additional questions about what computational architectures and algorithms can be used to most effectively capture, store, index, retrieve and merge these data, how multiple learning subtasks can be orchestrated in a larger system, and questions of computational tractability.

Considering this definition, machine learning requires samples of information of a system that represent the systems characteristics. These samples are referred to as "features", as defined by Bishop in [8]:

> In machine learning and pattern recognition, a feature is an individual measurable property or characteristic of a phenomenon being observed.

Therefore an initial step in every machine learning process is the extraction of features. The process of feature extraction does not only require the data to be present, but also to be preprocessed and combined in a meaningful way. The term "meaningful" in this context is depending on the machine learning method, applied to the data. The relevant approaches for this thesis are covered in Section 5.2. In general there are three different types of machine learning:

1. supervised,
2. unsupervised and
3. reinforcement learning.

Mohri et al. describe supervised learning in [4] as follows:

> Supervised learning is the machine learning task of inferring a function from labeled training data.

This means that, given a data-set, the desired output of the machine learning algorithm is a function that best fits the given data and can be used to predict a systems behavior for new samples. One example for such a method is the so-called k-Nearst-Neighbor (KNN) algorithm, which is explained in further detail in Section 5.2.2. In contrast to supervised learning, unsupervised learning is provided with non-labeled data. The aim of unsupervised learning methods is not to approximate a function but to extract an underlying structure of the data-set. In this context, structure could be clusters or similar attributes. An example algorithm for unsupervised learning is the so-called k-Means algorithm as described in more detail in Section 5.2.1. Eventually, reinforcement learning uses a feedback-function instead of labeled data in order to provide the algorithm with feedback regarding the learning process as defined by Sutton in [24]:

> Reinforcement learning is the learning of a mapping from situations to actions so as to maximize a scalar reward or reinforcement signal. The learner is not told which action to take, as in most forms of machine learning, but instead must discover which actions yield the highest reward by trying them.

5.2. Methods

5.2.1. K-Means

The k-Means algorithm, also referred to as Lloyd's algorithm is an unsupervised, iterative machine learning algorithm. In [27] Wu describes the k-Means algorithm as clustering algorithm that can be applied to reveal K non-overlapping clusters within data-sets. The clusters are represented by their centroids. For this, initially the number of clusters (K) has to be submitted. Next an initial distribution of the cluster centers has to be set. In the project of this thesis the k-Means implementation of the scikit-learn[1] tool is used. This tool utilizes the so-called "k-means++" method, as proposed by Arthur and Vassilvitskii, for the initialization step in [5]. The authors propose to chose the first cluster centroid randomly among all samples of the data set. The remaining cluster centroids are initialized based on the distance of the remaining samples to the already generated centroids. For this, all but the first centroids are picked, among the sample points based on the probability, given by Equation 5.1. In this equation, $D(x)$ denotes the distance of the sample x from the closest centroid that has already been chosen and X is the set of all samples.

$$\frac{D(x^2)}{\sum_{x \in X} D(x^2)} \tag{5.1}$$

Once the cluster centroids are initialized, the samples of the data-set are assigned to these clusters based on Equation 5.2. In this equation, c_i denotes the index of the chosen centroid μ for the sample x_i.

$$c_i = \underset{j}{\operatorname{argmin}} \ \|x_i - \mu_j\|^2 \tag{5.2}$$

Subsequently the centroids are updated, based on the assigned samples, according to Equation 5.3, which denotes the arithmetic mean value of all samples that belong to a cluster.

[1] `http://scikit-learn.org`

$$\mu_j = \frac{\sum_{i=1}^{m} \delta(c_i - j) x_i}{\sum_{i=1}^{m} \delta(c_i - j)}$$
$$\delta(x) = \begin{cases} 1 & x = 0 \\ 0 & otherwise \end{cases} \tag{5.3}$$

The algorithm then continues to reassign the samples and update the cluster centroids until the difference between two subsequent iterations is minimal or zero.

In general there is no guarantee to find the optimal solution when applying this algorithm. However due to a priori knowledge about the initializer and the properties of the feature space, the algorithm is a good choice for the use case in Section 6.3.4 and yields good results as exemplified in Figure 6.12.

5.2.2. K-Nearest-Neighbor

The k-Nearest-Neighbor algorithm is a supervised machine learning method that can be used for classification of data. Berman describes the k-nearest-neighbor algorithm in [7] as:

> A simple and popular classifier algorithm that assigns a class (in a preexisting classification) to an object whose class in unknown. [...] From a collection of data objects whose class is know, the algorithm computes the distances from the object of unknown class to k (a number chosen by the user) objects of known class.

It can therefore be utilized to assign a sample to already classified data, based on a distance metric. For this, initially a model is provided with already labeled data. Subsequently this model is provided with unlabeled data in the same feature space. The aim of the model is then to provide a label for the unlabeled sample. In the project of this thesis, the k-Nearest-Neighbor algorithm is used with a k of one in Section 6.3.5.

6. Project

Given a worldwide known multiplayer online battle arena (MOBA) game, analysis on the current continuous integration tactics have been performed. A combination of various different approaches aim to minimize the probability of a faulty product. Many of the implemented tactics are automated processes that should minimize the need of human resources. However, the analysis have revealed that most of the critical server-side issues are discovered, using tactics that require a large amount of human interaction. These tactics involve many recurring tasks as well as some properties of human behavior. As these processes are very time-consuming, a lot of human resources are constantly invested to repeat mostly the same procedures frequently. In addition, most parts of these processes could be very well automatized and therefore executed by machines. Thereby the staff would be relieved and the working time could be spent more efficiently.

The aim of this project is to generate a system that combines the continuous integration process with artificial intelligence. It should take over the tasks that currently require a lot of human interaction, to relieve the staff of the company and increase the quality of the continuous integration process.

6.1. Setup

The following section shall first provide a quick overview over the structure of the system itself. Following the initial situation of the continuous integration process is compared to the desired situation after the project is finished.

6.1.1. Overview

The system under test consist of a server and a client. The server is responsible for physics calculations, processing of user-inputs, matchmaking and other aspects that are critical to the game result. It incorporates several applications that handle different aspects of the game, such as session handling, database management and physics calculations. In addition to the server, the game features a client that is mainly responsible for providing the user with visual feedback and sending the input data to the server. Both instances, the server as well as the client, create a variety of logs that help the programmers to identify problems when they are encountered.

6.1.2. Continuous Integration: Initial Situation

Currently, most server-side problems are encountered by human players during so-called "playtests". During these tests the staff is playing the game with the aim of revealing bugs. The game contains a large amount of players that can play against each other simultaneously. This amount of players exceeds the number of staff that is available. To counteract the limited number of staff, a system is available that can generate computer-controlled players (bots). In the initial implementation, it is possible to generate a desired number of bots. Every bot is capable of joining a game and uses an adapted version of the client program to interact with the server and engage in the game. In this adapted version of the client, the bots receive information which is critical to the gameplay, just like in the client for the human players. However it is possible to start the bot client without any visual feedback, making it possible to launch several bot players on the same machine without running out of resources. For this the bots can be configured using launcher scripts. These launcher scripts are written in python and contain any information that is needed before the game starts. Information that is part of the launcher script is therefore:

- The number of bots to create,
- the session which has to be joined
- and any further parameters that can only be altered before the battle has started.

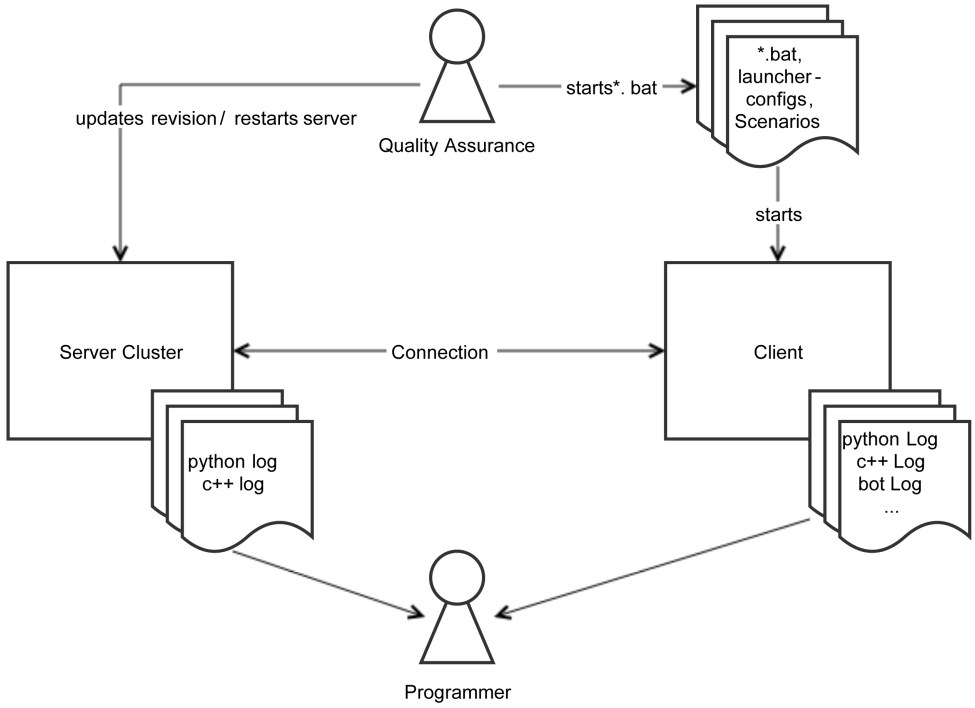


Figure 6.1.: In the initial situation playtests are supervised by the QA department. Sessions are generated by human players while the bots join these sessions to compensate for the insufficient number of staff.

In addition to the launcher script, the QA-department is provided with several batch files. These batch files start the bot launcher with predefined launcher scripts and server configurations. The process of playing a session including bots is depicted in Figure 6.1.

During playtests replays are created and the server, as well as the client, generate different log-files that help the programmers to identify problems. These log-files are collected manually and have to be scanned by the programmers to identify the problem. This processes of log-data creation using play-tests and manual scanning of log-files are very time-consuming. Some parts are recurring and can therefore be automatized. Other parts, such as the scanning of the log-files, may be more productive by using a more appropriate representation of the data. The aim of this project is to come up with a solution to these problems

to enhance the quality of the continuous integration process.

6.1.3. Continuous Integration: Desired Situation

In order to improve the continuous integration process a new system should be created that automatizes most of the previously mentioned processes. This new system shall be configurable with a single script. The execution of this script shall update the server to the desired branch and revision. Also a desired number of bots that play on the server shall be started and play session all by themselves. Compared to the initial method, using this system should not need any human interaction. At the end of every game, the generated logs should be gathered at a central location and processed such that they can be queried easily. Additionally, a report containing information about encountered problems should be sent to the corresponding experts. This automated process shall be repeated over night with various different configurations. This should provide for a better coverage regarding the functionality of the game and therefor increase the range of errors that can be revealed by the system. The desired new system configuration is depicted in Figure 6.2.

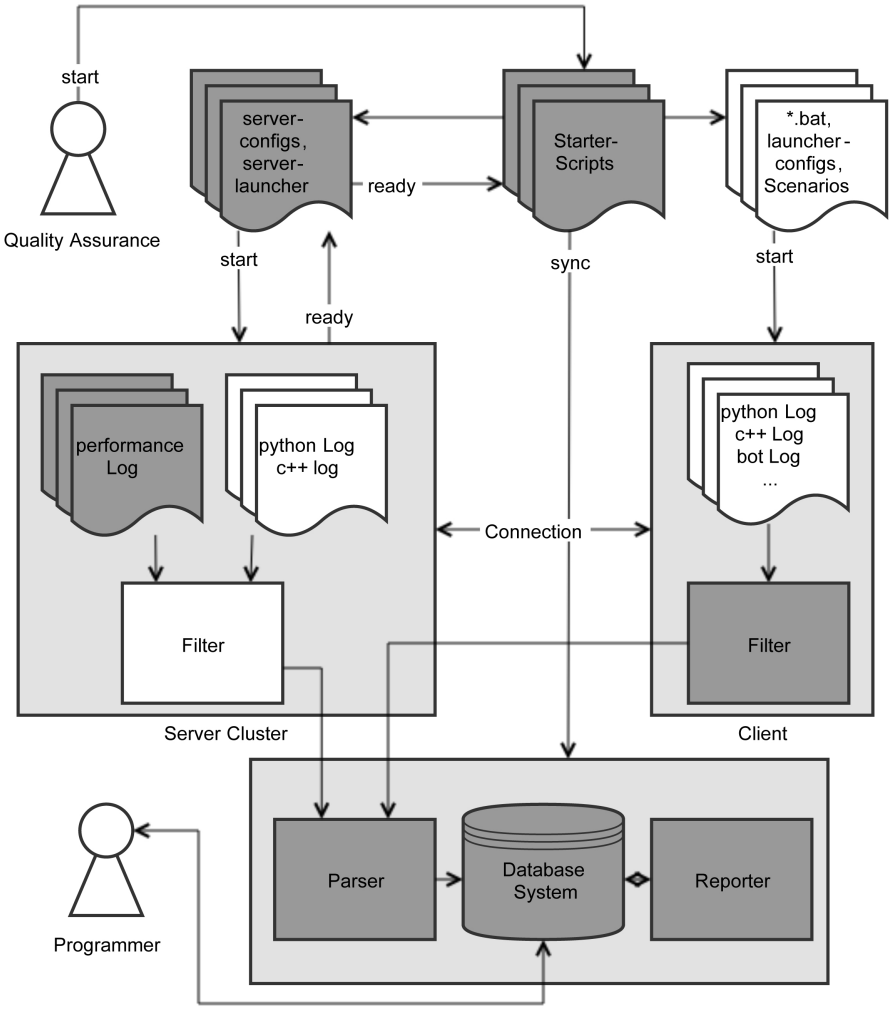


Figure 6.2.: In the desired situation playtests can be executed autonomously. Sessions may be generated by bots and several playtests can be played subsequently. Eventually, these tests shall be executed overnight so the programmers can analyze the generated log-data in the morning. (White components are components which are already existing at the start of the project. Dark gray are the components that need to be integrated.)

6.2. Automatic Testing and Logging

In this section, the methods used to implement every component depicted in Figure 6.2 are covered from bottom to top.

6.2.1. Database System

The main goal of the database is to deliver an easy to query system for the programmers. Additionally it should provide tools that simplify the analysis of log-information. A research on such systems revealed an open-source software collection called Elastic Stack by Elastic[1]. This software collection includes a system called Logstash[2] which can gather and process data from different sources, so it can easily be stored in a database-system. As database system, another product of the same company called Elasticsearch[3] is used. Eventually the Elastic Stack also features a graphical user interface called Kibana,[4] which provides a web-service that can be accessed directly via a browser. In Kibana the programmers can view the raw data or filter and connect it in a meaningful way. Also visualizations can be created directly in the web-interface to reveal hidden data. The Elastic Stack is integrated into the system as depicted in Figure 6.3.

For this project, the Elastic Stack was deployed as Docker Container. Instructions regarding the installations of the Elastic Stack as Docker Container can be found online[5].

In Elasticsearch the data can be organized using so-called index patterns. Therefore every entry is assigned an index value upon reception. An example value for the index-entry used in this project would be the string "playtest01-2017.03.01". This example identifies data belonging to the first playtest on the first of march in the year 2017. The user can therefore define an index pattern with this string and subsequently filter all data from this specific playtest.

[1] https://www.elastic.co
[2] https://www.elastic.co/products/logstash
[3] https://www.elastic.co/products/elasticsearch
[4] https://www.elastic.co/products/kibana
[5] https://elk-docker.readthedocs.io/

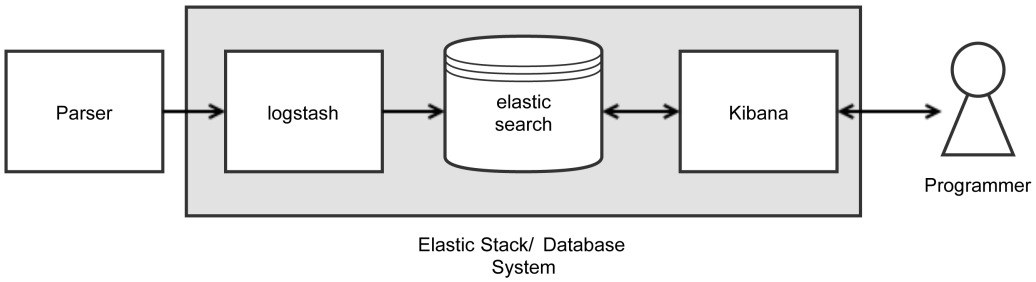


Figure 6.3.: Integrating the Elastic Stack into the project: The parser sends data to Logstash via TCP. Logstash is configured to accept TCP-data and populate the database in Elasticsearch with the new information. Kibana is used as graphical user interface to query Elasticsearch.

6.2.2. Log Parser

The Log Parser is a program, written in Python, that was created for this project to forward the log data to logstash. It exploits the servers capability of printing log data to the console in real-time. For this the server provides a program that can be executed from an ssh shell. Therefore the Log Parser is capable of connecting to given servers using an ssl-client such as putty[6]. After having connected to the desired server it analyzes every line that is printed by the server to the shell. In this project the log-message data is always formatted as depicted in Table 6.1.

Weekday	Day of Month	Month	Year	Time	App	Severity	Message

Table 6.1.: Composition of log data printed to the shell by the server after separation by single or multiple white-spaces. The slightly gray colored fields are merged to a single time-stamp entry.

The first five fields of Table 6.1 are merged to a single time-stamp, utilizing the "strptime"-method of the datetime[7] module of Python. In this project, the server contains a variety of apps that work together. To distinguish between these apps in the logs, the source app for every log-entry is added to the "App"-field. The severity of the message (for example Warning or Error) as well as the message itself are both added to the corresponding fields.

While most log-entries are single lines terminated by a new-line character, a few entries may be separated into several lines. Also these multi-line entries do not have to appear in sequential order, but can be interrupted by another entry that does not belong to the multi-line entry itself. Therefore multi-line log-entries are extended with meta-data. All separated lines that belong to the same multi-line entry are extended with a prefix. The prefix consists of the combination of a continuation character and an identifier. The continuation character ("/", "|" or "\") indicates whether the current line of the message is the start, the end or an intermediate line of the log-message. The identifier is a random hexadecimal number with six digits. An example multi-line message is depicted in Listing 6.1. For processing multi-line entries a python module was

[6] https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html
[7] https://docs.python.org/2/library/datetime.html

created. It contains a class for buffering the data until it is complete, as well as functions for the extraction of the identifiers and the removal of the prefix.

```
1 /D6E9F1 [TRACEBACK EXT]
2 |D6E9F1   File "some/path/someFile.py", line 60, in FunctionName
3 |D6E9F1     locals: {'Var1': {'Var2': 'Val1', 'Var3': 'Val2'}}
4 |D6E9F1   Error: (1054, 'unkmown identifier "test"')
5 \D6E9F1 [/TRACEBACK EXT]
```

Listing 6.1: Example for a multi-line message. Every message that is separated into several lines is extended with a prefix to indicate the affiliation. Additionally a continuation character indicates whether the line is the start, the end or an intermediate line.

In order to reveal messages that occur with high frequency, all messages are processed before they are sent to Elasticsearch. The aim of this step is to revert the variable data of the log-message to a format string. This is accomplished using regular expressions. Therefore two messages of the same source with different variables are equivalent after the processing step. As the messages can be very long and therefore hard to reference, every message is assigned a unique identifier. This identifier is a consecutive number that is incremented with every new message. In this context a new message is every processed message that does not have an identifier assigned. Functions for processing of the messages, as well as the maintenance of the mapping from identifier to processed messages is included in an own python module. This module handles the assignment as well as generation of new UIDs. It utilizes the "search" API of Elasticsearch as described in Section 6.2.3. The module provides methods to convert message to their reduced representation and assign UIDs. For the assignment of UIDs the module queries the Elasticsearch server for a UID for the given message in order to keep the data consistent. In case there is no UID for the message available on the server yet, it is allocated and reused otherwise. An example for the use of this module is shown in Listing 6.2

```
# given uid mapping uids
uids={
      (1,"unique message with integer %d"),
      (2,"another unique message with no variables"),
      (3,"message with string '%s'"),
}
# analyze the new message
newMsg = "This message has 1 new string 'test string'"
uniqueMsg = toUnique(newMsg)
newUid = assignUID(uniqueMsg, uids)
```

Listing 6.2: Example of message processing and identifier assignment. The unique message in this example would evaluate to the string "This message has %d new string '%s'" and would be assigned the identifier 4.

After having decoded the received data, the parser converts it to json as described in Listing 6.3. The log-data is sent to Logstash after this process. Logstash is configured to accept json data via TCP and extract the "@idx"-field as the index-name of the package. This field describes to which index the package belongs with respect to the chosen index pattern, as described previously. The remaining fields include data that shall simplify the process of debugging for the programmers. Every single field will be added to the Elasticsearch database by Logstash automatically upon reception.

```
{
  '@idx': index,
  '@kind': 'LOG',
  '@time': timestamp,
  '@level': severity,
  '@source': app,
  '@message': msg,
  '@uid': uid,
  '@message_reduced': msgReduced,
  '@novelty': isNew,
  '@server': serverName
}
```

Listing 6.3: The log-data as sent to Logstash. The data is collected by the parser, decoded and formatted in json as depicted above.

In addition to log-data the parser also queries the servers load in regular time

intervals. The server features a program that can print the load of every server application to the console. Therefore the parser uses an ssh-client to query this data regularly, just as for the log-data. This data is queried in a separate thread that is executed simultaneously to the main thread which processes the log-data. This separate thread executes the command to retrieve the load in regular intervals. This periodic requesting of data is accomplished in a bash script as depicted in Listing 6.4.

```
1 while true do
2   date '+%a %d %b %Y %H:%M:%S.0'
3   cluster-control | tr '\\n' '~'
4   sleep 5
5   done
```

Listing 6.4: Bash script to periodically query the servers load. It operates in two phases. The First phase is to query the time of the sample, the second phase is to query the sample itself. After both phases it waits for a given number of seconds (5 seconds in this case) until it repeats the process.

The bash script consists of three commands, wrapped in a loop. The first command is used to retrieve the current time-stamp of the server. The second command retrieves the servers load. Unfortunately the load for each server application is entered in a separate line. In order to get the whole data as single entry, some transformation has to be performed. In this case the new lines are replaced by tilde characters. Subsequently the parser separates the load of each application by using the tilde as delimiter. The last command of the bash script puts the process to sleep for a given amount of seconds.
In the parsers polling thread for the servers load, the information about the load is therefore collected in two stages. First the time is retrieved, converted to a time-stamp and buffered. In the second stage, the load information is retrieved and decoded. The decoded information is subsequently completed with the previously buffered time-stamp.

The load data of every server app is extracted and sent to Logstash separately, formatted as depicted in Listing 6.5. Most of the fields are equivalent to the log-data fields.

```
{
  '@idx': index,
  '@kind': 'LOAD',
  '@time': timestamp,
  '@app': app,
  '@cpu': cpu,
  '@mem': mem,
  '@server': serverName
}
```

Listing 6.5: The load-data as sent to Logstash. The data is collected by the parser, decoded and formatted in json as depicted above.

The parser supports two different operation modes. The modes are called "direct" and "intermediate" mode. These modes differ from each other regarding the route of the packages from the source server to the Elastic Stack server. The mode can be altered using the "feederMode"-attribute of the playtest configuration as described in Section 6.2.4, Table 6.2. In direct mode the parser is launched directly on the source server and is configured to send the parsed json-messages to Elastic Stack server as depicted in Figure 6.4. The intermediate mode was introduced for the case that source- and Elastic Stack server do not share a common sub-net or the parser cannot be installed on the source server. In intermediate mode the parser is launched on a machine that can connect to source- and Elastic Stack server. In this mode the parser acts as proxy server. It queries the data from the source server using ssh and forwards it to Elastic Stack using TCP. The intermediate mode is illustrated in Figure 6.5. Further information about the system execution can be found in section 6.2.4 where the collaboration of all system components is elucidated in detail.

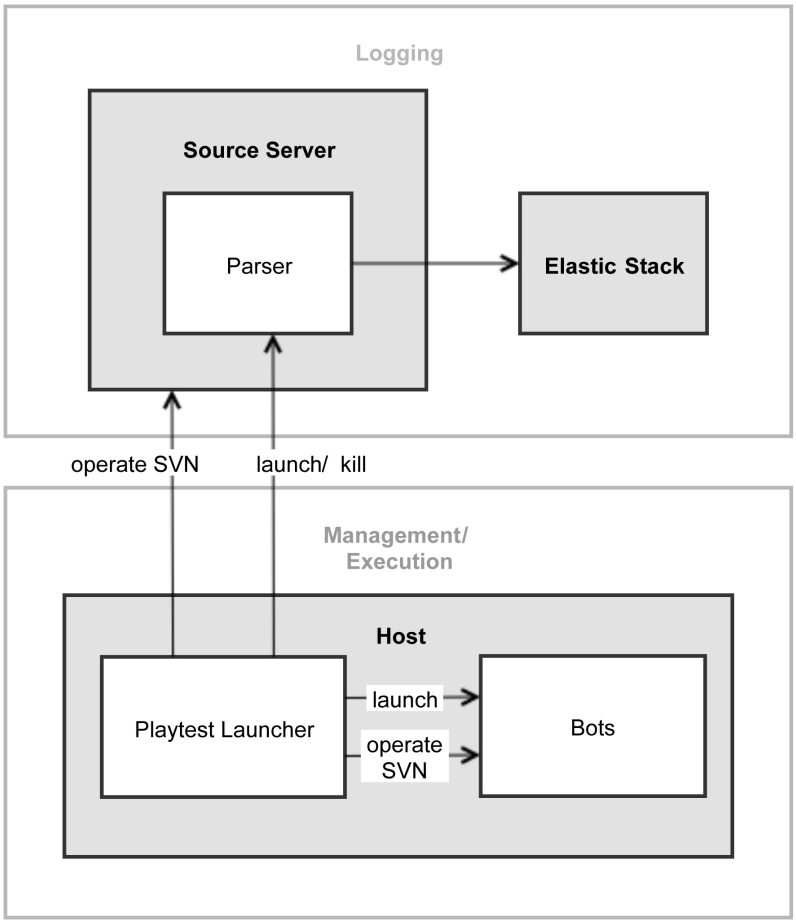


Figure 6.4.: This figure illustrates the execution of the system with the parser running in direct mode. In this mode, the parser is executed on the source server and the data is sent directly to Elastic Stack.

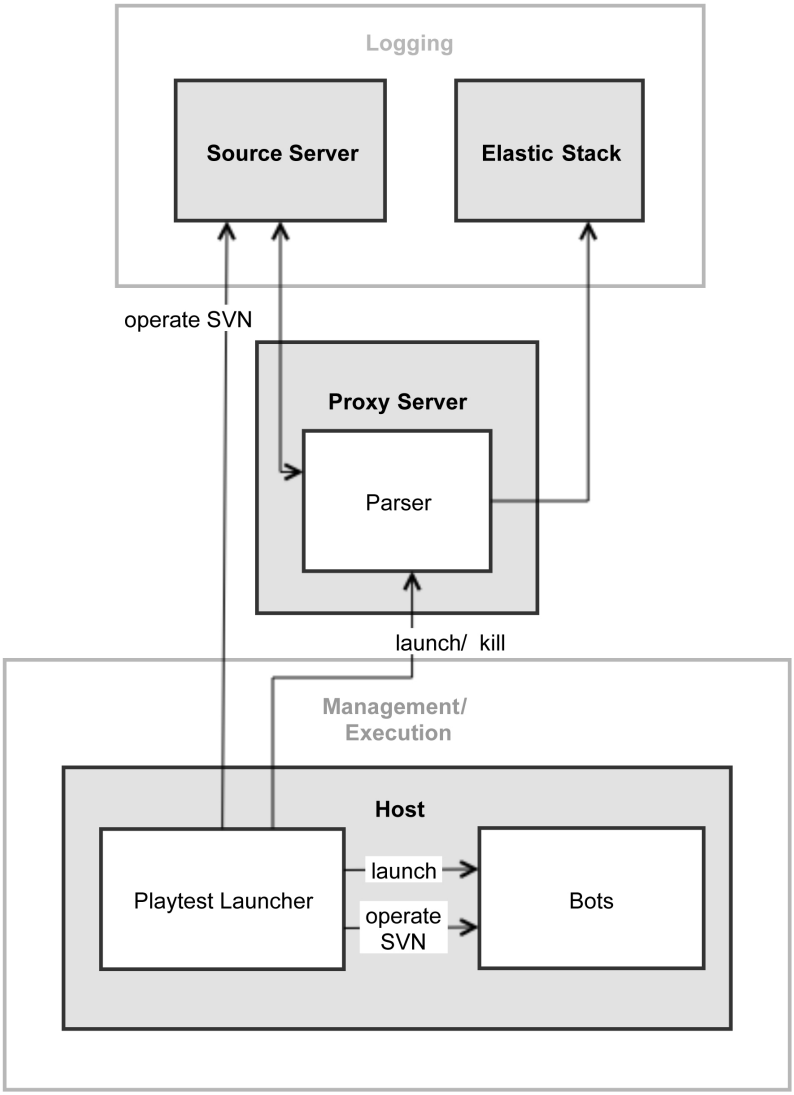


Figure 6.5.: This figure illustrates the execution of the system with the parser running in intermediate mode. In this mode, the parser is executed on a proxy server that can connect to the source server and Elastic Stack. The parser connects to the source server using ssh to query the logging data and forwards the processed data as TCP-messages to Elastic Stack.

Client Log Data

The aspects discussed in this chapter so far apply to processing the server-side log data. Unlike the server, the client does not feature a filter functionality for log data out of the box. Neither does processing the client log-data require an ssh-connection as the client is executed on the same machine as the test system. Further information regarding the system execution can be found in Section 6.2.4. In contrast to the server log parser, the client log parser can be executed after the playtest as the data is stored locally to a log-file and can easily be separated by playtests after the execution. In general this is also possible for the server log data, but rather complex to achieve. For this reason the server logs are processed online. Nevertheless the client log parser supports both, online as well as offline processing. When executed in online mode, the parser polls for change in the log-file in regular intervals and forwards the data to Logstash while the game is executed. In offline-mode, the client parser expects the path to the log-file as parameter. The mechanics of the client log parser are similar to the server log parser. A connection to Logstash is established, messages are parsed, UIDs for messages are generated, multi-line entries are merged and the extracted data is sent to Logstash. For this some manager classes are shared with the server log parser and adopt to slight differences. Other components such as the manager for multi-line entries have been completely recreated for the client due to the massive differences in implementation between server and client logs. In contrast to the server log parser, the client log parser does not send system statistics to the server in regular intervals. Also, the filter functionality is directly implemented in the parser. This is possible as the severity of a message is part of every log-entry in the client log. Therefore the parser can simply check the value of this severity entry after having parsed the log line and skip it in case it is not of interest. In addition to the message kinds, referenced in Listing 6.3, the client logs introduce the new value "CLIENT_LOG" for the "@kind"-field.

6.2.3. Reporter

The purpose of the reporter is to generate a report for a given playtest that summarizes the log-data for the playtest. It shall provide an overview over the encountered log-messages and the average server load during the playtest. An example server report can be found in the Appendix A.2.1. It includes branch, as well as revision of the source server and the index of the playtest data. Additionally it provides an overview over the encountered number of log-messages of different severity levels. These are separated into total and unique messages. The total message count includes the total number of messages encountered at a given severity level. The unique message count is the number of messages of same source, which is the number of distinct messages after having replaced variable data with format strings. The server load data of the playtest is averaged and also included in the report. Eventually the distinct messages are listed, sorted by severity and total appearance count.

In order to generate a report, the reporter requires the index pattern of the desired test as input. Optionally it can send the generated report as email. For this the sender log-in credentials for the email server, as well as the receiver email addresses have to be submitted. The reporter can also filter the messages, based on predefined unique messages. As a result, these messages will be removed from the main view and appended to the end of the report in a separate section. However the filtered messages are still included in the total count of the overview. In order to apply filters to the report, a file including the filters has to be submitted as parameter. This file is a simple plain-text file which includes the reduced version of the messages that have to be filtered, separated by newline-characters.

Server Queries

The reporter utilizes the RESTful APIs of the Elasticsearch server in order to query the logging data. For this, several queries have been created that shall be discussed in further detail.

The first entry of the report that requires a server query is the name of the server that was used for a specific playtest. In order to get this information the reporter sends the query defined in Listing 6.6 to the Elasticsearch server. This

query utilizes the "search" API of Elasticsearch which may be used to search for data. This query evaluates to a single server name entry of any message received for the given index pattern. This is sufficient as the server name is part of every message received.

```
1 http://ES_SERVER_IP:PORT/INDEX_PATTERN_NAME/_search?_source=
    @server&size=1
```

Listing 6.6: This is the query the reporter uses in order to find out which server was used for a specific playtest. As the server name is part of every message, it is sufficient to query any message of the given index pattern for the server field entry.

The queries for branch and revision work similarly to the server name query. The difference is that the branch and revision information is not included in the server load update messages but only in log messages. Therefore this constrained has to be added as depicted in Listing 6.7.

```
1 http://ES_SERVER_IP:PORT/INDEX_PATTERN_NAME/_search?_source=
    @branch&q=@kind:LOG&size=1
```

Listing 6.7: This query is used by the reporter to request the branch of a given playtest.

The total number of entries is retrieved using the "count" API instead of the "search" API of Elasticsearch. The reporter queries the total number of messages for every severity level separately. The query in Listing 6.8 shows the query for the total number of messages with the severity level "ERROR".

```
1 http://ES_SERVER_IP:PORT/INDEX_PATTERN_NAME/_count?q=@level:ERROR
```

Listing 6.8: This is the Elasticsearch query to retrieve the number of log messages with the severity level "ERROR".

In order to retrieve the log messages for every severity level, the query from Listing 6.9 is used. This query provides the reporter with the message as well as the reduced message and the UID of every entry for the given severity level. Unfortunately Elasticsearch requires a size parameter which defines the maximum number of results as it cannot work with an arbitrarily large result set. Therefore the size parameter has to be chosen sufficiently high. The server load data is queried the same way.

```
1 http://ES_SERVER_IP:PORT/INDEX_PATTERN_NAME/_search?pretty=true&
    _source=@message,@message_reduced,@uid&q=@level:ERROR&size
    =10000
```

Listing 6.9: This is the Elasticsearch query to retrieve message, reduced message and UID of all messages with the severity level "ERROR" for a given index pattern.

Report Generation

After all necessary data has been queried, the report can be generated. At this stage the reporter utilizes pythons list comprehension capabilities to further analyze and process the data. Eventually the reporter generates two report versions with the same data. One plain-text-version and an html-version. The content of these reports are equivalent however the html-version also includes links to Kibana and is formatted using html-tags to be more readable. After the reports have been generated, both versions are saved to non-volatile memory and -if so desired by the user- sent via email. The email is formatted to always show the html version if this is supported by the email client of the receiver. Otherwise the plain-text version will be presented. This is realized utilizing the python SMTP module and modules from the "mime"-package. The email always includes the html-version of the report as attachment.

Client Report

All aspects discussed in this chapter so far apply to report generation for server logs. A report can also be generated for client logs. For this the previously mentioned aspects also apply to client logs, however there are a few additions. Unlike the report for server logs, the client log report can be generated without Elastic Search. For this, the reporter can be provided with a path to a log-file. The reporter can utilize the parser to parse the log-data into a local data-structure and use this data as source for the report. In addition to that the client report can also be generated using data from Elastic Search as source. In this case the reporter has to be provided with the desired index, as for server logs. In contrast to server log data, the parser for client logs does not provide any information about the clients load as described in Section 6.2.2. Therefore

this section of the report is not included in client reports. An example client report is added to the Appendix A.2.2

6.2.4. Playtest Launcher

The purpose of the playtest launcher is to automatize the execution of playtests. Therefore it is the heart of the system that manages all processes involved in a playtest. Following, these processes are listed in execution order and steps two to nine are repeated for every playtest in the playtest setup:

1. Loading the playtest modalities
2. Setup of the SVN repositories on server and client
3. Launching of server
4. Starting the parser
5. Launching the bots (Client) to play the game
6. Stopping the bots once the game is over
7. Stopping the parser
8. Launching the reporter
9. Reverting the previously changed SVN settings

The playtest launcher expects a configuration file for the playtests to be submitted as parameter. This configuration file is a json-file that describes the modalities of the desired playtests which are to be executed. It is possible to define as many tests as desired. An example configuration file is added to the Appendix A.1.

The Tables 6.2, 6.3, 6.4 and 6.5 shall briefly describe the possible entries of the configuration file. Each of these tables describes the setup of the components of the playtest. The first component is the basic playtest setup, describing the index name, relevant severity levels for the parser, the operation mode of the parser and similar adjustments. The second component is the server setup including the name of the server as well as optional SVN settings. The bot setup is the third component and mainly describes how the game shall be played, including number of players, how to join and other important bot settings. The last component is the optional reporter setup. If this setup is not provided the reporter will simply generate a report and save it to non-volatile memory.

Many of the table entries cover subjects that have been covered in previous sections. The remaining entries shall now be covered in more detail.

Field Name	Datatype	Compulsory	Default Value	Description
indexName	string		(queried on demand)	Name of the index for the playtest in Elasticsearch
localEcho	boolean		true	Print log-data to local console
createIndexPattern	boolean		false	If true, the system will automatically generate an index-pattern in Kibana for the playtest
noLogging	boolean		false	skip logging for this playtest
logSeverities	list		["CRITICAL", "ERROR", "WARNING"]	Log entries of these severity level that will be forwarded by the server. Have to be entries from following list: ["TRACE", "DEBUG", "INFO", "NOTICE", "WARNING", "ERROR", "CRITICAL", "HACK"]
feederMode	string		-	The parsers operation mode
serverSetup	dictionary	yes	-	(see below)
botSetup	dictionary	yes	-	(see below)
reportSetup	dictionary		(report will not be sent)	(see below)

Table 6.2.: This table describes the fields available to configure the basic playtest setup.

For convenience, the system supports automatic generation of index patterns in Kibana. This function will generate an index pattern in Kibana, that matches the exact index of the messages for the given playtest. This is accomplished by sending an http-PUT-request to the URI "$http://ES\_SERVER\_IP:PORT/.kibana/index-pattern/INDEX\_PATTERN$". Additionally the server expects the name of the time-field which identifies the order of the packages within the messages. This is submitted in the data section of the package as illustrated in Listing 6.10.

```
1 {
2   "title" : INDEX_PATTERN,
3   "timeFieldName": "@time"
4 }
```

Listing 6.10: This is the content of the data field of the http-PUT-request to generate a new index-pattern in Kibana.

Server Setup

The server setup dictionary is used to identify the server which is to be used for the playtest, as well as the desired SVN settings of the servers repository. All available servers are stored in a global, hard-coded list in python. Every entry

Field Name	Datatype	Compulsory	Default Value	Description
name	string	yes	-	Name of the server as defined in constants / hosts file
branch	string		None	Path to the desired SVN branch
revision	int		None	The desired SVN revision
skipRollback	boolean		false	The playtest launcher will not reset the servers SVN settings to initial settings if this flag is set

Table 6.3.: This table describes the fields available to configure the server setup for a playtest. It shall provide the playtest launcher with the server name and the desired settings regarding the SVN repository.

of this list includes the servers name, which is used to identify the server, as well as the IP-address and the absolute path to the directory where the parser is resided on the server. In the configuration it is therefore enough to submit the desired servers name, any additional information is retrieved from the list. Sometimes it may be required to check the status of a certain revision in a specific branch. Or it may even be desired to check two different SVN-states throughout the course of a single nightly test-phase. For this case, the user may specify the desired SVN-settings using the revision and branch field of the server setup dictionary in the configuration. The playtest launcher will then update the servers SVN-repository to the desired configuration, prior to launching the game. The playtest launcher utilizes the putty link program to connect to the server and execute SVN-commands on it. The default behavior of the system is to revert the SVN-settings to the initial state in case changes were made. If this is not desired, the "skipRollback"-flag has to be set in the configuration.

Bot Setup

In order to understand the customization options of the bots, the bot-system itself shall briefly be described. The basic framework that is used in this project is mainly comprised of server, as well as client and bot code. However, the bots can be launched in two different modes. One of these modes allows a massive amount of bots being launched in a simplified environment. This environment only features a GUI with a list of all the bots and some meta-data like log-output and states. In this mode, none of the bots features a graphical

Field Name	Datatype	Compulsory	Default Value	Description
svnPath	string	yes	-	Absolute path to the SVN root where the bots are resided
botRepoPath	string	yes	-	Absolute path, where the bots scenarios are located
launcherBatFile	string	yes	-	Bat-file name for launcher
branch	string		None	Desired SVN branch for bots
revision	integer		None	Desired SVN revision for bots
scenarionName	string	yes	-	Name of the scenario which is to be executed for this playtest
numTeams	integer		2	Generate bots for team 1 (1) or both teams (2)
numPlayers	integer		12	Number of bot players per team
forceTankClass	string		None	Tank class that will be used by the bots
ignoreQueueLimits	boolean		False	If set to true, the bots will ignore the limits for tank classes when using the queue
isHumanSupervised	boolean		False	If set to false, the first bot will open a lobby
useClientPlayer	boolean		False	If set to true, the first bot will launch a client with GUI
creator	string		”default”	Creator of the lobby which is to be joined (only used when isHumanSupervised is true)
loginName	string		”@bot%team_%index”	Desired login name for the bots
loginPWD	string		””	Desired login password for the bots
hangarMode	string		”TRAINING”	QUEUE or TRAINING (joining method)
serverMode	string		”DEBUG”	DEBUG, or SPA (for login and joining)
laneTeamLimit	integer		13	Number of bots per lane per team
tanksPerBot	integer		2	Minimum number of vehicles per bot
initialWaitTimeAtt	integer		0	Initial wait time for attackers before they start playing
initialWaitTimeDef	integer		0	Initial wait time for defenders before they start playing
space	string		„“	The Navmesh which is to be loaded. Blank entry means automatically detect

Table 6.4.: This table contains the fields available to configure the bot setup for a playtest. It also includes the SVN settings for the client used by the bots to connect to the server and play the game.

Field Name	Datatype	Compulsory	Default Value	Description
filterFile	string		None	Path to the desired filter file
recipients	list		None	a list containing the recipients of the report

Table 6.5.: This table describes the fields available to configure the reporter. This configuration part is fully optional and will lead to a report being saved to non-volatile memory and not sent via mail if not present.

representation of the game. These bots only communicate with the server and do not use the client, which is used by human players to visualize the game and handle user-inputs. The other mode is called "client"-mode which provides the bot with a standard client as it is used in release versions of the game. In this mode the bot interacts with the game just like a human player. A comparison of these modes is depicted in Figure 6.6.

(a) This is the default bot mode, where the bot directly communicates with the server.

(b) This is the client bot mode, where the bot uses a client instance to communicate with the server just like human players do.

Figure 6.6.: This figure illustrates the two different operation modes of the bot system.

The bots are typically controlled by the bot launcher. This launcher is the heart of the bot system. It generates all bot instances and manages the bot execution. The launchers technical parameters can be customized using command-line-arguments. Using these arguments, parameters like default behavior in case of an exception, as well as loading of geometry data and other settings can be submitted. Therefore it was decided to wrap the start of the launcher in a bat-file and make these parameters relative to the playtest configuration. The launchers technical configuration can be customized using the "launcherBatFile"-entry of the bot-configuration. It also defines the desired scenario as explained in Section 6.3.1.

In addition to the command line arguments, the launcher expects a configuration file, containing gameplay-relevant information. This file is formatted in json and includes setup data for the bot execution. It is structured as dictionary which includes meta-data, such as:

- Login credentials of the bots;
- Number of bots;
- Bot execution mode (default or client);
- Desired scenario;
- Any other settings, needed for the desired scenario.

The setup of the bots requires a few customization steps that produce and load different configuration files. These steps are illustrated in Figure 6.7.

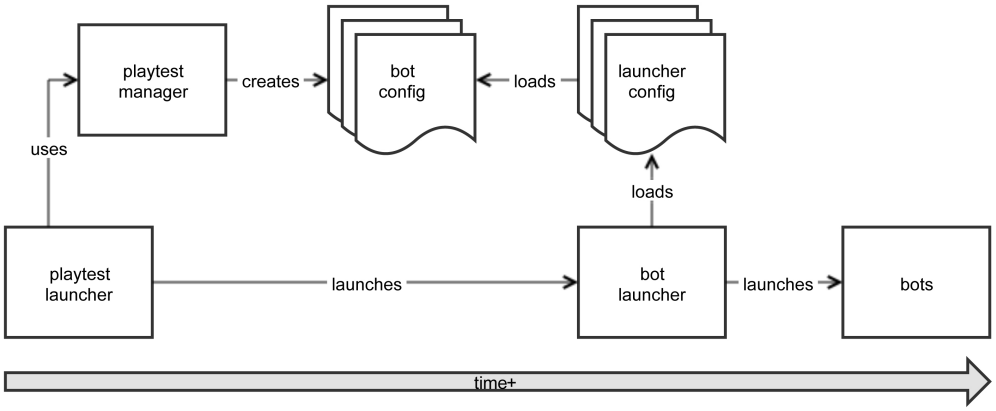


Figure 6.7.: This figure illustrates how the bots are customized and executed. The playtest launcher utilizes the playtest manager to create a configuration file for the bots. The bot-launcher, the heart of the bots, loads the launcher configuration which will load the previously created bot configuration to start the bots as described in playtest configuration file.

Scenario is a term that has been mentioned several times in context of bot execution in this section. The scenario is the section of the bot code that defines the bots behavior. The scenario starts when the bot has launched and is connected to the server and ends as soon as the connection to the server is closed. This means that this code segment can be used to emulate the players

behavior. A very basic pseudo code example of a scenario is defined in Listing 6.11. The scenario code is covered in more detail in Section 6.3.

```
1  WHILE inLobby DO:
2    connectToSessionFromPlayer(playerName)
3    waitUntilGameStartedOrSessionClosed()
4
5  WHILE inGame DO:
6    WHILE objectiveNotReached(currentObjective) DO:
7      navigateTowardsObjective(currentObjective)
8    WHILE NOT missionAccomplished(currentObjective) DO:
9      executeMissionForObjective(currentObjective)
10     currentObjective++
```

Listing 6.11: This pseudo-code exemplifies how the bots operate. They stay in the lobby and wait until the game starts. Once the game has started they accomplish one task after the other until the game was won or lost.

Just like the server, the clients SVN-repository can also be altered for a playtest. For this the client configuration features the same fields as the server configuration. The difference in terms of execution is, that the bots are always executed on the local machine. This is also the machine where the playtest launcher is executed, which means that no remote connections are used to update the SVN repository in this case.

Another option of the bot configuration that is not specifically designed for scenarios and has not yet been discussed is the "isHumanSupervised" flag. This flag can be utilized to use the system for playtests that are organized by human players. This means if the staff wants to test the game and also needs bots to participate and gather logging data, this flag can be set. If this flag is set, the bots will join the session of the human player, specified in the "creator" field of the configuration. The remaining behavior of the bots will be the same as for automatized playtests.

Report Setup

The reporter setup is fully optional. If it is skipped, the default behavior of the reporter is to generate a report and save it to disk. However, it is possible to apply filters to the report as mentioned in Section 6.2.3. This requires the "filterFile" field of the reporter configuration to be filled with the path to the

filter file. Another option of the reporter is to have the report sent via email. For this the configuration offers the "recipients" field which can be filled with a list of receiver email addresses.

Execution

The execution of the playtests is managed by the playtest launcher and can be divided into three phases:

1. Setup phase,
2. Execution phase and
3. Postprocessing phase

These three phases are depicted in Figures 6.8, 6.9 and 6.10 which are part of a single sequence diagram depicting the execution of a single playtest. The three phases of the playtest launcher shall now be explained in more detail.

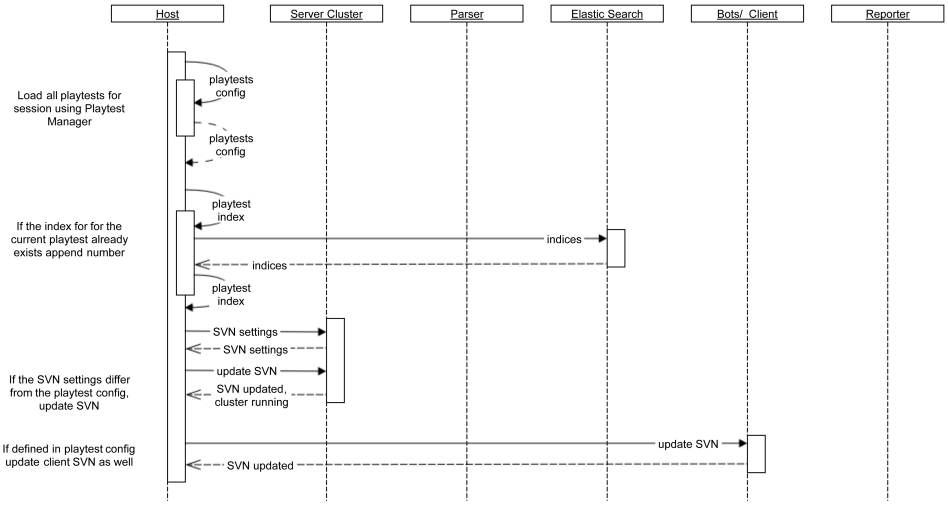


Figure 6.8.: This Sequence diagram illustrates the setup phase of the playtest launcher. This includes loading the playtest configuration from file, utilizing the playtest manager, as well as the setup of the SVN repositories of server and client.

During the setup Phase, as depicted in Figure 6.8, the playtest launcher gathers the modalities of the playtest(s) by loading the playtest(s) configuration file(s).

This configuration file includes a desired index name for referencing in Elastic Search, as described in Section 6.2.4. In case this parameter is not submitted the user will be queried for a name in the console. As this requires human interaction, setting the index-name parameter in the playtest configuration is highly advised, but not mandatory. In order to allow for the same index name on multiple days, Logstash is configured to append the current date to the index-name upon the reception of a package. Therefore, as an example, the playtest with the index-name "playtest" will be changed to "playtest-2017.03.01". In addition to that, the playtest launcher detects if data with the given index is already present for the current date. For this it utilizes the "_cat" API of Elastic Search by sending the query defined in Listing 6.12. In case the index has already been used, the playtest launcher will append a consecutive number of two digits to the index name. Therefore the example index-name "playtest" will be changed to "playtest01" locally. This index-name will then be changed to "playtest01-2017.03.01" upon package reception by Logstash.

```
1 http://ES_SERVER_IP:PORT/_cat/indices?v
```

Listing 6.12: This is the query the playtest launcher sends to Kibana in order to retrieve the list of indices. As a result, the playtest launcher will receive a message from Kibana including all available indices.

After having generated a unique index-name, the playtest launcher queries the server for the current SVN branch and revision using an ssh connection to the server. In case the playtest specifies a desired branch and revision that mismatch the current data, the servers SVN repository is updated. For this, initially the servers services are stopped. Next the current SVN-data of the server is backed locally in a plaintext file. This backup guarantees that no SVN setup data is lost in case the update fails. Also the backup is used to revert the settings after the playtest is over. Eventually the servers SVN repository is updated and the services are started again. All functions regarding the SVN management of the server have been organized in a server SVN manager class in a separate Python module.

A client SVN manager class is also available which performs similar tasks for the client. The client is not always running and thus does not need to be stopped in advanced. Also the client does not require an ssh connection as per design the client is always installed on the same system as the playtest launcher. It was also decided to skip the SVN rollback option for the client.

In case any critical steps fail during any of the playtest launchers phases, the playtest launcher will log the problem, skip the test where the problem occurred, revert the SVN settings if changes have been applied and continue with the next playtest. After the setup phase, the system is ready to be executed with the desired configuration.

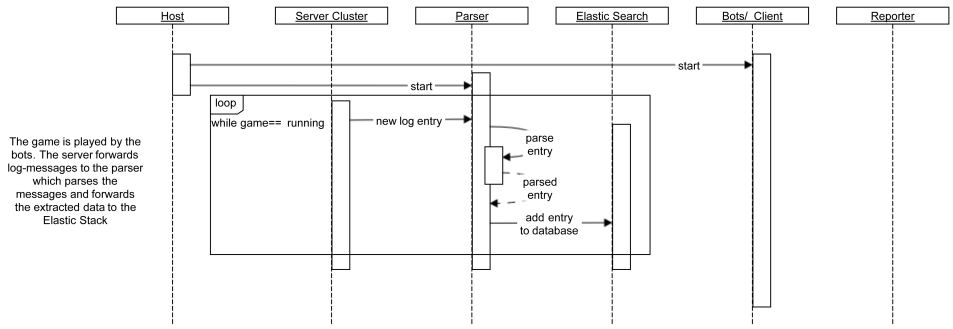


Figure 6.9.: This Sequence diagram illustrates the execution phase of the playtest launcher. The execution of the playtest starts with the launch of the bots. The bots connect to the server and permanently exchange data with it. The connection between server and client is not included in the diagram as it does not directly effect the functionality of the system. During the execution the server creates log-data that is forwarded to the parser. The parser processes the data and forwards it to the Elastic Stack server.

The execution phase starts with the launch of the bots as sub-process and is depicted in figure 6.9. In this context, launching the bots is equivalent to launching the bot-launcher, which is responsible for managing bot-instances, as descibed in Figure 6.7. At this point the system adapts to the values of the attributes "isHumanSupervised" and "useClientPlayer", described in Section 6.2.4. The values of these attributes describe whether the system is to be run autonomously and if there shall be bots using a visual feedback. If the system is used for human supervised playtests, the bots will be configured to join the session of the player defined in the "creator" of the bot setup. *The implementation of the basic bot system, including session handling has been provided in advance and is not part of this project. The aim of this project regarding the bot system is to manage the configuration and improve the quality of the bots. More details regarding the bot system can be found in Section 6.3.*

In case the playtest shall be executed autonomously, the system configures the first bot to start a session and the others to join. In this case, the value of the "creator"-entry will be ignored. If the "useClientPlayer"-flag is set, the system configures the first bot of each team to start a client. The rest will be default bots without a client. This way, the system tests all available components including server, client and bot code. The bots are configured as described in Section 6.2.4 in Figure 6.7, using a configuration file that is creates by the playtest manager module at the very beginning of the playtest during setup phase.

The server parser is launched almost simultaneously to the bots in order to catch log-messages as early as possible. The server parser is launched on the target server, using the command in Listing 6.13.

```
1 exec python PATH_TO_PARSER MODE INDEX_NAME SEVERITIES
    SERVER_BRANCH SERVER_REVISION SERVER_NAME ECHO_MODE
```

Listing 6.13: This is the command used to launch the server parser. The parser is always executed on either the source server or a proxy server. Therefore it expects the meta-data about the playtest as parameters.

The target machine can either be a proxy server or the source server of the log data, depending on the entry of the attribute "feederMode" described in Section 6.2.4. As the parser is not launched on the local system, it has to be installed on the target machine and therefore expects the playtests meta-data as parameter. A Python script was created to install the system on a target machine automatically in order to keep the installation folder consistent and ensure the systems functionality.

The execution phase represents the testing of the system. In this phase the game is played and log-messages are created by server, client and bots. During this phase the server logs are parsed by the parser in real-time and forwarded to Logstash as described in Section 6.2.2. If a client exception occurs at any time during the execution phase, the bot-launcher is configured to quit and return a non-zero value. Otherwise the bot-launcher terminates returning zero.

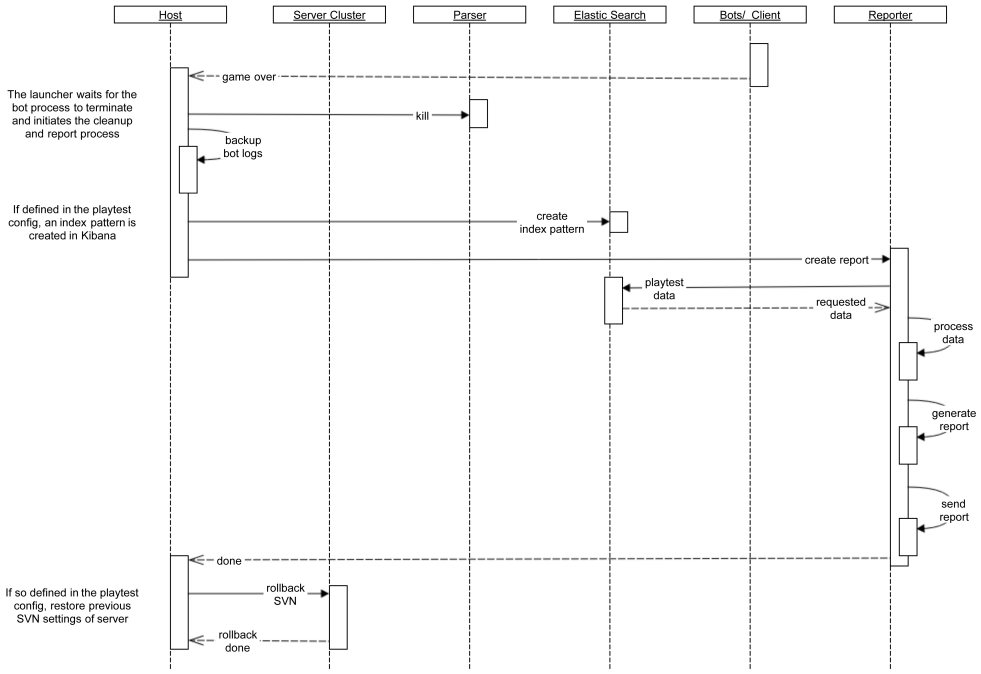


Figure 6.10.: This Sequence diagram illustrates the post-processing phase of the playtest launcher. Once the bot process terminates the parser is killed and the produced log-files are backed up to disk. Subsequently an index pattern matching the playtest is created in Kibana and the reporter is triggered to generate and send a report. Finally the servers SVN repository is revert to the state before the playtest was started.

Once the game is over, the testing is done and the system initiates the post-processing phase as depicted in the sequence diagram in Figure 6.10. The start of this phase is defined by the check of the return value of the bot-launcher. This value indicates whether the test was successful. In case a critical error was encountered, either bots, client or server log will reflect this error. In addition, a message is added at the top of the report that a critical error was encountered. Independent of the result of the test, the playtest launcher cancels the execution of the parser. This is accomplished by executing the bash-command, described in Listing 6.14 on the machine where the parser is executed. This implies that the parser has to be executed on a Linux machine.

```
pkill -f /feeder.py
```

Listing 6.14: Bash command to kill the parser. The parser is often referred to as "feeder" in this project as it feeds Logstash with information.

Once the parser has been killed, the log data of the bots and client are backed up for further analysis by the programmers if necessary. Next the playtest launcher creates an index pattern in Kibana that matches exactly the index of the current playtest. This step is skipped if it is not specifically desired according to the playtest configuration. In case the index pattern is created, the following report will contain a link to Kibana with the data filtered according to this index-pattern, otherwise it will contain a link to index-pattern creation site of Kibana.

After these steps the playtest launcher utilizes the reporter in order to create and send a report for server, as well as client. As already discussed in Section 6.2.3, the server reporter queries the data directly from Elastic Search, while the client reporter uses the local logging data. After having gathered the data it is processed and inserted in a report template. If any receiver addresses for the report have been defined in the playtest configuration, the reporter will automatically send the report to the receivers in an email. For this, the reporter requires an SMTP-server to connect to and use as email gateway. The SMTP-server that is used in this project requires user log-in data in order to be allowed to send emails. The playtest launcher therefore expects email credentials as optional parameters. At the very beginning, the launcher loads all playtests and checks for the need of sending emails. In case no email credentials have been submitted and any of the defined playtests in the configuration requires emails to be sent, the credentials will be queried at the beginning of the playtest launcher as part of the setup phase.

6.3. Bots

As mentioned in previous Sections, the implementation of the basic bot framework is not part of this project. The aim of the bots in this context is to emulate human behavior as closely as possible. This is important as the game is usually played by humans. Therefore if the bot behavior is too different from human behavior, the probability of leaving errors unrevealed by bots and being found by humans is high. The aim of this project regarding the bots is to adapt the behavior of the bots to emulate human behavior as closely as possible without having to recreate the whole bot-framework. In this section, the relevant parts of the basic bot framework will be covered. Subsequently the measures taken to enhance the behavior are explained.

6.3.1. Basic Bot Framework

The basic bot framework allows for an arbitrary number of bots that is only limited by the computational power of the workstation where the bots are executed. The system distinguishes between default and client bots as illustrated in Figure 6.6. Default bots are simplified versions of client bots that do not feature any graphical feedback of the game. This highly reduced the computational requirements of the bots. However graphical feedback can be delivered on a per-bot-base. This requires the bot to launch in a different mode and can therefore not be changed at run-time.

The behavior of the bots can be altered in code-snippets that are referred to as scenarios. The scenario is the high level part of the bot system. All the management regarding memory, connections, scheduling is delivered by the framework. The framework supports multi-threading and suggests to create separate threads for session handling and in-game behavior. Therefore every bot scenario can be separated into three threads:

1. The main thread,
2. session thread and the
3. behavior thread.

The main scenario thread is called once the bot has connected to the server. The target server and log-in data, such as the behavior of other non-programmable aspects can be defined in the launcher configuration as described in Section 6.2.4. Once the bot has been connected, the scenario starts and it is up to the bot programmer to define the behavior of the bot. Therefore in the main thread usually the session and behavior threads are generated.

In the session thread any preparations for the game as well as joining of desired sessions are implemented. This part of the code is usually very repetitive and does not require certain aspects of human behavior.

The most important part of the scenario for this project is the behavior thread. In this thread the in-game bot behavior can be customized and human-like behavior is desired.

The bot framework allows for an arbitrary number of scenarios. The desired scenario is loaded by the bot-launcher upon start-up. This is how different bot-behaviors are implemented in this project. The desired scenario is part of the "launcherBatFile" in the bot configuration as described in Section 6.2.4. Therefore, various bat-files for different behaviors can be supplied.

The rest of this section describes how the behavior thread is adapted to create bots that emulate human behavior.

6.3.2. Creation of a Control System

The initial step to enhance the bot quality is to create a system that matches the character of the game. The game is a multi-player online battle arena game. Therefore it features several entities that roam in a virtual world and can manipulate various parameters. Due to the core mechanic of the game, any action by any player may directly effect the player, as well as the surrounding players. This means that the actions taken by players are often a function of previous events as denoted by Equation 6.1.

$$A_{t+1} = f(\vec{e}) \tag{6.1}$$

In Equation 6.1, the next action A_{t+1} depends on the event vector $\vec{e}$. In addition to that, every player has unique properties that may even change throughout the course of the session. Therefore the actions are also depending on the current properties p of the players, as defined in Equation 6.2.

$$A_{t+1} = f(p, \vec{e}) \tag{6.2}$$

If the combination of properties and events is combined to transitions $\vec{T}$ and action is defined as a state S the resulting equations are 6.3. These equations model a finite state machine (FSM) which seems to be a suitable candidate as central control instance for every individual bot.

$$\begin{aligned} \vec{T} &= (\vec{p}, \vec{e}) \\ S &= A \\ S_{t+1} &= f(S_t, T_t) \end{aligned} \tag{6.3}$$

In the game a transition is always a change in context such as sudden loss of health, appearance of enemies, reaching objectives and similar things. Examples for a state would be navigating or hiding. The FSM uses the combination of these states and transitions to control the behavior of a bot as exemplified in Figure 6.11.

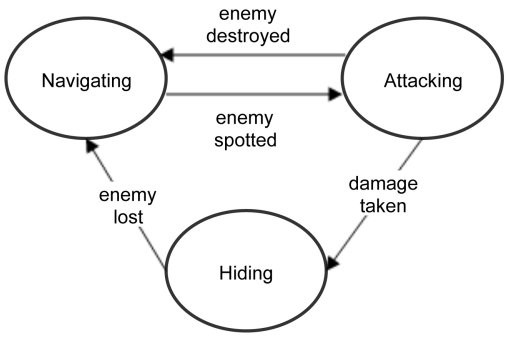


Figure 6.11.: This is an example of a very basic finite state machine to control bots. The blobs denote states which include scripted behavior. The arrows denote transitions, which are checked at regular intervals during the execution of a state.

The use of a FSM allows for a highly responsive behavior as the conditions of the players are checked in regular intervals.

6.3.3. Creating States

States are a collection of simple actions that form a behavior. In order to navigate, a bot has to load a navmesh, decide for a destination, calculate a path and control its avatar in the game to finally reach the destination. All these actions can be combined into a navigation-state. In practice it has shown that the basic solutions for states result in very poor performance compared to a human. This is attributable to the mechanics of the game.

Game Mechanics

The game consists of two teams playing against each other. Team one has to capture objectives while team two has to prevent team one from capturing the objectives. Team one wins as soon as all objectives are captured and team two if the time runs out and there are non-captured objectives left. Both teams also win if the enemy team is destroyed. Both teams have the option to navigate through a virtual map while trying to no be spotted by enemies. In addition every player has the option to shoot at enemies. A successful hit is largely depending on the hit-point on the enemies avatar. Therefore it is crucial for the bots performance to have access to a list of vulnerable points in order to get a well shooting performance. In addition the position of a player is a crucial property depending on the state of the game, especially regarding the position of enemies and allies. A good position is defined as a spot where enemies can be spotted and shot while not revealing the own position to the enemies. These two characteristics of the game have been chosen to be the main target of the project in context of bot behavior improvements.

6.3.4. Extracting Weak-Points for Bots

There are several possible approaches of how to adopt to the previously mentioned characteristics. Regarding the weak-points of the avatars the quickest

and simplest solution is to augment every avatar with meta-data, readable by the bots. Throughout the course of the project, access to a database, including data of millions of games, has been granted. As this database also includes a collection of damage data it was decided to apply machine learning on this data to extract the weak-points, in relation to the avatars. The damage includes following data that is used for the weak-point extraction:

1. The avatar type,
2. amount of damage taken,
3. the part of the avatar effected by the damage and
4. the position of the impact, relative to this part.

This implies that the avatar consists of different parts. These parts do all have certain properties regarding the absorption of damage. For this reason only the weak-points of the two most visible parts are extracted in this project. However, the code is designed to be able to extract weak-points for all parts. This however has a negative effect on the result. In order to extract the weak-points, first the damage data is filtered by actual damage. Any data below a certain threshold is ignored. After having filtered out the less influencing data, the rest of the data is prepared for analysis by a k-Means-Algorithm. The k-Means-Algorithm is an algorithm that can be used for clustering of data as described in Section 5.2.1. For this the algorithm is provided with all the impact points for an avatar part as feature set. In addition the number of clusters have to be submitted. As described in Section 5.2.1, the scikit-learn tool uses k-means++ for initialization. As this method initially distributes the cluster centroids at maximum distance from each other, a value of four for the number of clusters is chosen in order to get one weak point in every cardinal direction. With this amount of clusters, the distinct weak-points per avatar part are usually separated in every direction, relative to the center. Therefore there is one weak-point in front, one at the back and one at each side. For verification, the resulting data is visualized in a figure after the weak-point calculation. Therefore the relative transformations of the single parts in relation to the root of the avatar have to be loaded from an XML-file. After this, the transformations can be applied to the data and printed in 3D-space for verification. An example result is depicted in Figure 6.12.

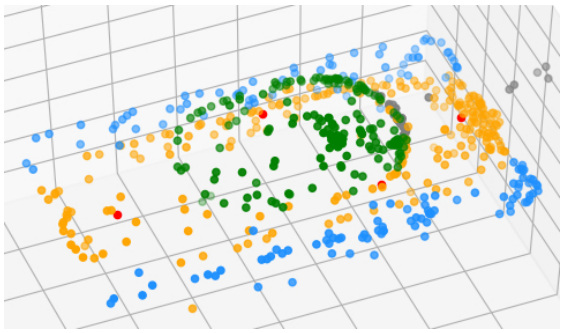


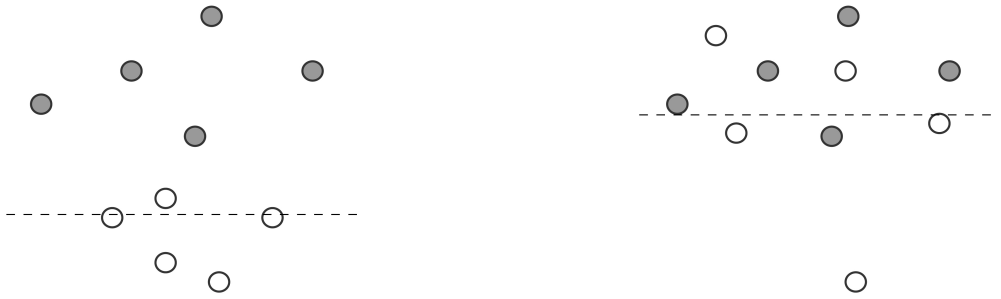
Figure 6.12.: This is an example visualization for the result of a weak-point extraction. The different colored points all denote a hit-point, except for the red ones. Every color corresponds to a different avatar part. The red points in this visualization denote the extracted weak-points for the orange avatar part.

The weak-point extraction is executed for every single avatar type the bot may encounter during the game. After the extraction the resulting weak-points are written to a CSV-file. The content of such a weak-point CSV-file can be found in the Appendix B.1. The name of the file corresponds to the avatar ID in-game. In-game the bot has access to this avatar type id and can therefore exploit the weak-points using these files.

6.3.5. Extracting Navigation Targets for Bots

In the most straight forward implementation of the bots, the navigation targets are formed by the primary objectives. These are accessible points of interest delivered by the game. However these points only take the primary objectives into account and do not reflect the current situation of the game, which is a crucial parameter for success. This is why the straight forward approach of simply approaching one point of interest after the other does not perform well in terms of emulating intelligence. If both teams use this approach, the result is that all bots gather in a small area fighting each other. The game is not designed to be played this way and human players would always prefer a different approach as the design of the game suggests to act differently. In this section a countermeasure for this undesired behavior is discussed.

In addition to the primary objectives the players are provided with a front-line. This front-line represents the distribution of the players on the map as illustrated in Figure 6.13.



(a) This figure depicts the front-line as dashed line. The players of different team are denoted by differently shaded dots.

(b) This figure illustrates how the front-line changes in case one team moves closer into the zone of the other team in comparison to the figure on the left.

Figure 6.13.: This figure illustrates how the front-line works. It directly reflects the distribution of the players in the virtual map.

As illustrated in Figure 6.13, the front-line maps the positions of all players to a single float value. It therefore compresses the multidimensional data in $\mathbb{R}^{3*num\_players}$ to a one-dimensional representation in $\mathbb{R}$. This compressed data is available to the bots at any time and due to it's low complexity and availability forms a good measure for the current state of the game.

In order to exploit the front-line data in a meaningful manner, machine learning is applied. The aim of the learning process is to obtain a model that maps the combination of current player- and front-line position to a trajectory. This trajectory is to be roamed by the navigation state until a point of interest or enemy is encountered. If one of these events occur the FSM switches to a more suitable state to handle this change in situation. A trajectory in this context is a path that a human player has chosen in the same, or a similar situation. In this context a situation is the combination of front-line position, player position and player team.

Obtaining Trajectories

In order to obtain possible trajectories, the replay-feature of the game is exploited. If desired, the game features the ability to create replays of every match. This replay can be used to export events of the replay to a plain-text-file. This event list contains information about the players and their position updates as well as the front-line position and miscellaneous other data. The data is sorted by a time-stamp denoting the ticks of the game. In order to maximize the quality of the result, only the replays of the best players are taken into account. The replay and event list contain the position updates of all players involved in a session. Therefore as an initial step, the identifier of the desired player is extracted. Using this ID as unique delimiter for position updates, the entries of the event list can be filtered. The remaining entries are parsed in order to gain the desired information. The player as well as the front-line data include a time and a position. For the front-line only one of the position entries is relevant. The player, as well as the front-line updates have to be put into relation with each other in order to allow for a successful training phase during the machine learning process. For this the time-stamp is used. The game however features a higher number of player position updates than front-line updates. Therefore as an initial step, the same front-line position is merged with multiple player positions. For this the player updates are extended with the front-line updates at minimum time-stamp distance. The index of the front-line update is chosen by solving the minimization problem from Equation 6.4. In this equation the parameter N denotes the number of distinct front-line updates while pt and ft are the time-stamps of the player and front-line updates.

$$\underset{0 \leq i < N}{\operatorname{argmin}} \|pt - ft_i\| \tag{6.4}$$

The resulting data is structured as exemplified in Table 6.6. The mapping, using the time-stamp is necessary in order to assign a front-line position to every player position.

After the data has been preprocessed as previously described it is separated into distinct trajectories. This is necessary, as the game also features a re-spawn feature that teleports the player to another position. This re-spawn exposes the player to a completely different situation and therefore it has to be processed

time-stamp player position	time-stamp front-line	player position	front-line position	interpolated front-line position
2	1	(-15.8, 22.3, 2.3)	20.0	20.0
5	1	(-15.3, 22.9, 2.2)	20.0	20.325
8	1	(-14.5, 23.5, 2)	20.0	20.975
9	16	(-14, 24.1, 2.1)	21.3	21.3
11	16	(-14.2, 25, 1.9)	21.3	22.52
...				

Table 6.6.: This table exemplifies how the data for the machine learning process for the navigation target determination is structured. The front-line position is linearly interpolated in order to compensate for the limited resolution.

as separate trajectory. As the motion of the avatars in the virtual world are continuous a re-spawn can be detected by a far distance traveled between two subsequent time-stamps. Therefore the data-set is scanned for distances that exceed a given threshold and split into separate trajectories when the threshold is surpassed. As a result an additional column is added to the data-set representing the trajectory identifier. In order to able to evaluate the result a visualization is created from the extracted data. An example visualization is depicted in Figure 6.14

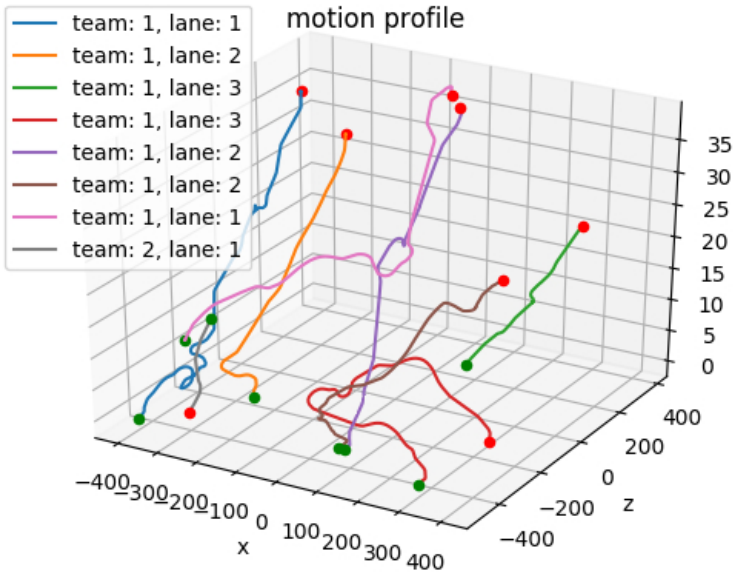


Figure 6.14.: This is an example visualization for the extracted trajectories of two different matches. The green dots mark the start of a trajectory and the red dots the end. The line in between marks the path traveled over time. The front-line data is not included in the graph.

In order to compensate for the low resolution of the front-line data, the position data of the front-line is linearly interpolated. The interpolation result corresponds to the last column of Table 6.6. The last attribute that is important but not included in the table is the so-called ”lane”. The game separates the virtual map into three lanes, each having an own front-line. The lane can simply be extracted from the front-line position and assigned an ID from one to three. An example file resulting from a trajectory extraction is depicted in Appendix B.2.

Training and Evaluating the Model

Using the k-nearest neighbor algorithm, described in Section 5.2.2, a model is fit to the trajectory data. For this the position of the player as well as the front-line position is used as feature-space. The data-source can either be a single file denoting a single match, or a folder including the trajectories extracted from a range of different matches. The algorithm is initialized with a k of one. Hence it delivers the closest sample of the training set to a given test sample. Therefore the bots can use the trained model to find an entry point to a trajectory that best represents the current situation in the game. Situation in this context denotes the position of the bot and the front-line.

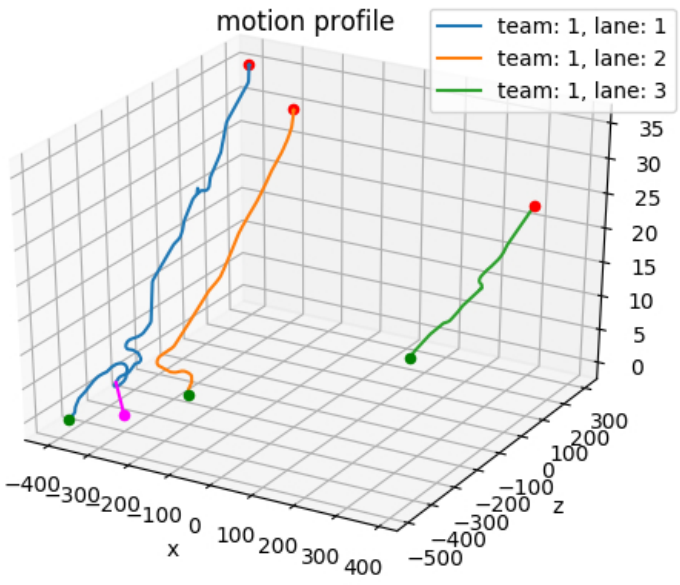


Figure 6.15.: This is an example visualization of the result of the KNN algorithm. The pink dot marks the sample position and the pink line connects the sample position to the chosen trajectory entry point.

6.4. Results

6.4.1. Test- and Logging System

The automatic test system has been deployed and tested. It has been configured to run the same test every night to check whether the changes, made during the day, have broken any functionality. For this it starts a session containing exclusively bots. During this session the log-data is uploaded to the Elastic Stack so it can be analyzed by the programmers when they enter the office in the morning. After the test has finished a report is sent to the programmers containing an overview over the results. An example for such a report is added to the Appendix A.2. The report contains a link to the overview page of Kibana with the data filtered to only display the data of the test, referenced in the report. Figure 6.16 depicts an example visualization used to overview the encountered logs.

6.4.2. Bot Improvements

The initial targeting system of the bots included a simple hard-coded offset in relation to the enemies avatars center. The bots were programmed to target this point and only shoot in case it was not covered by the static geometry of the virtual world. This resulted in bots not shooting at all in case this single hard-coded point was covered. The newly extracted weak-points offer a variety of target points which are unique to all avatars. This way the target point is not covered by static geometry, making the bots more aggressive when encountering enemies and therefore more "human-like". In addition to that, the target points are varied relative to the enemies transformation, making the currently most vulnerable part the main target of the bot. This corresponds to the shooting behavior of good players.

Testing the new navigation target system of the bots yielded a variety of new insights on the movement behavior of the players. As an initial step the trajectories without interpolated front-line have been tested. As a result, most of the bots targeted the closest point to a trajectory, ignoring the front-line parameter. This is attributable to the low resolution of the front-line data

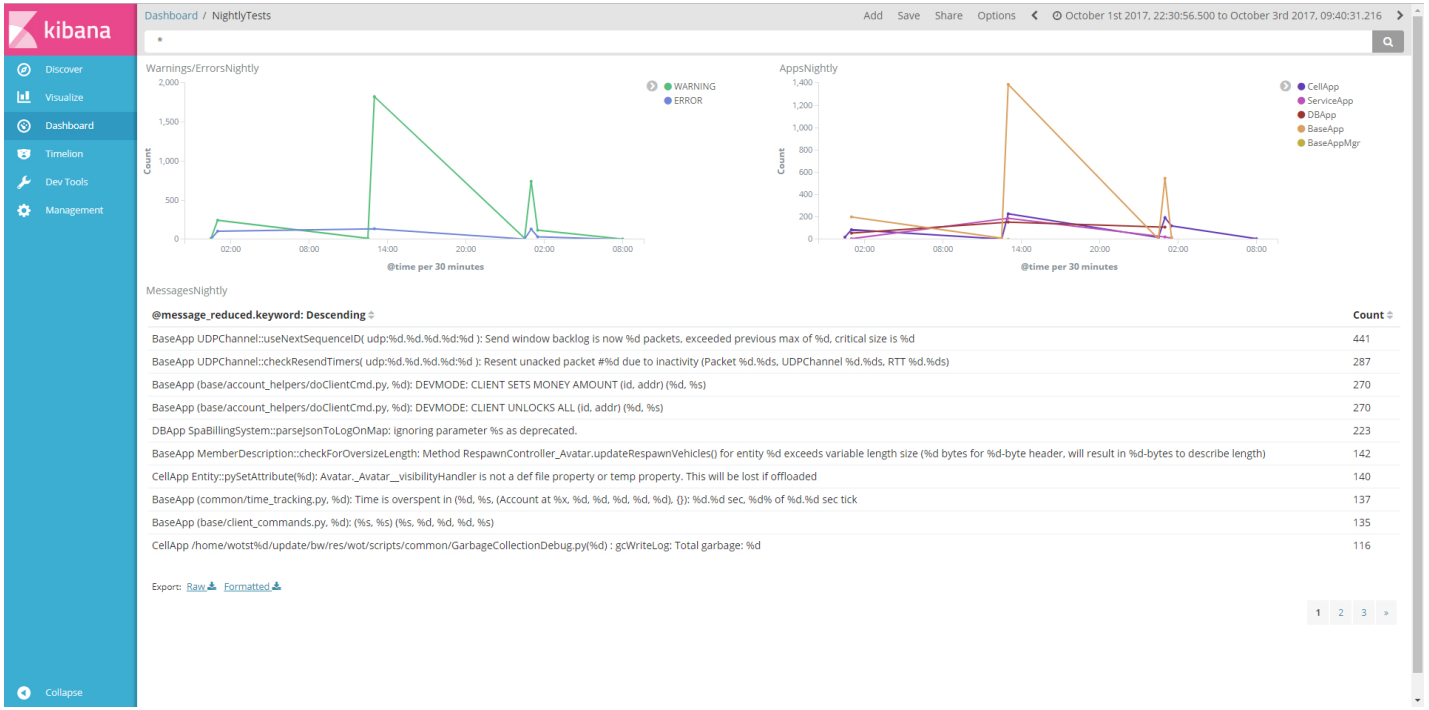
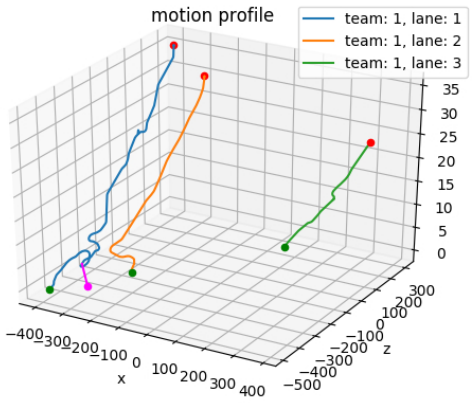
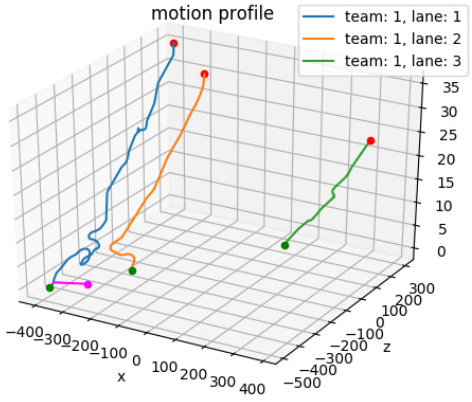


Figure 6.16.: This Figure depicts an example visualization in Kibana. In the top left it visualized the number of messages, filtered and categorized by the severity levels "ERROR" and "WARNING". On the top right is shows the number of messages, clustered by the source server application. The bottom provides an overview of the encountered messages, sorted descending by the number of occurrence.

and the reason why the data is interpolated. Due to the interpolation a more suitable entry point to the trajectories is chosen. This means that bots tend to proceed closer into the enemies territory in case the front-line is ahead. On the other head the bots tend to retreat if they are alone in enemy territory as depicted in Figures 6.17 and 6.18.

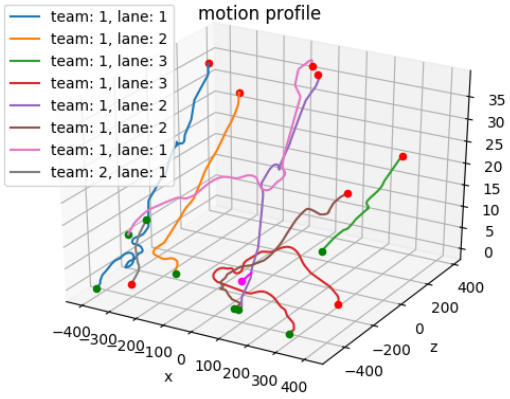


(a) This figure depicts the chosen entry point and trajectory for the given test sample in pink. In addition to the position, the sample also includes a front-line distance which is not depicted in this diagram. In this example the front-line distance is set to 300. Therefore the entry point is chosen in direction of the enemy territory.

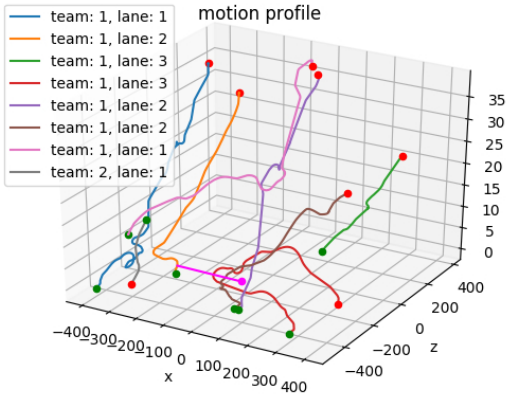


(b) In comparison to the figure above this sample has a front-line distance of -200. Therefore the entry point is towards the territory of the own team.

Figure 6.17.: This figure illustrates the influence of the front-line when choosing an entry point for the same trajectory. The higher the front-line distance value, the closer the front-line is to the territory of team two. Vice versa, the smaller the value of the front-line, the closer it is to the territory of team one. The test sample is assigned to team one.



(a) The sample point in this figure is adjusted to belong to lane two, team one and has a front-line distance of zero.



(b) In comparison to the example in the figure above, one the front-line distance has change to a value of 200.

Figure 6.18.: This example illustrates that the front-line information can even influence the choice of the trajectory. The red trajectory belongs to the wrong team while the brown and violet trajectories do not feature any samples that represent the desired front-line data adequately. This means that the data of the red and brown trajectories depict players that have moved deeply into enemy territory by themselves.

Also the algorithm correctly filters trajectories for the wrong team and lane as depicted in Figures 6.19 and 6.20. These results look promising in testing, however the approach does not perform well in-game.

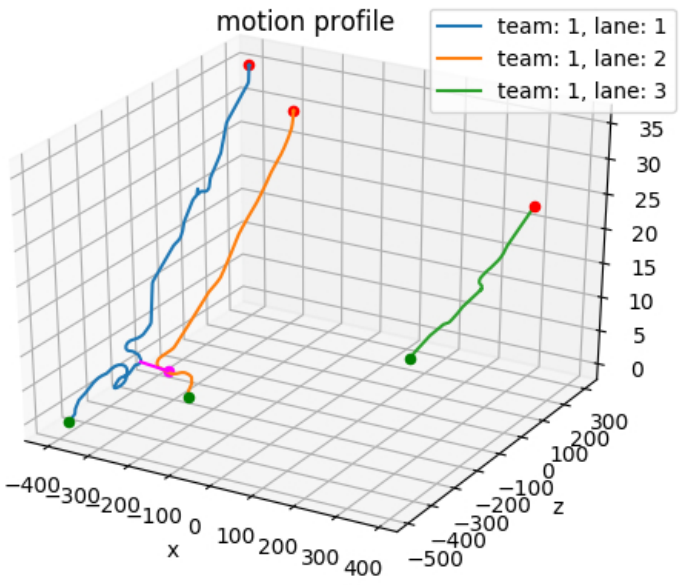


Figure 6.19.: This figure shows the influence of the lane parameter. Due to the perfect match in position, the obvious choice would be to decide for the orange trajectory. However, the sample requires a team one trajectory, therefore, the algorithm decides for the blue trajectory.

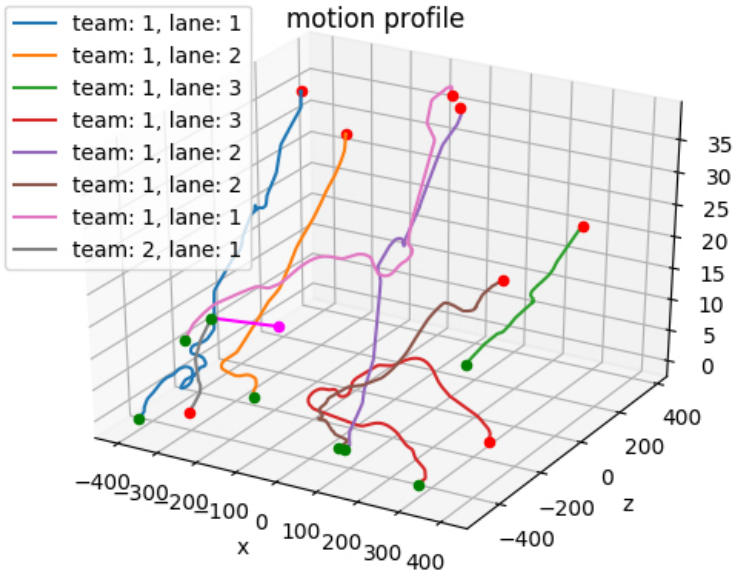


Figure 6.20.: In this figure, the influence of the team parameter is shown. The pink sample point is parametrized to belong to team two and lane one. Only the gray trajectory fulfills these requirements in this example.

The bad in-game result is attributable to a variety of facts. One problem in the implementation of this trajectory approach is the reactivity to events. Ones the bot has reached a trajectory it will roam this trajectory until the end is reached. However this trajectory represents a path a player has chosen in another session. The entry point might have been a good choice but the course of the trajectory is most likely a bad choice for the current session. Due to the low update interval of the front-line in-game this disadvantage is hard to compensate for. In addition the trajectories do not take the avatar kind into account, which is not optimal as different avatars have different attributes regarding speed and armor. As these attributes turned out to be a crucial factor for human players regarding the choice of the path, the behavior of the bots strongly deviate from human behavior in this context. Initially the bots tend to group by trajectory as the initial start point of the bots are close to each other. Due to a lag of

data, the entry point and trajectory are therefore equivalent to many of the bots, resulting in the bots forming groups.

7. Conclusion

Automatic software-testing is a key-component to success. Especially for large-scale software projects utilizing continuous integration and delivery it is an inevitable part of the delivery process. Automatic software tests can be used to reveal critical errors and ensure that the software works as expected. Automatic tests are usually predefined tests that should be repeated in regular intervals to ensure the functionality of the software. If these tests are not automatized they have to be at least executed by humans to some extend. However humans are more likely to introducing errors, especially when repeating the same process over and over. This is why setting up an automatic test environment might be time-consuming at the beginning but is definitely worth the effort.

The Elastic Stack combines data storage, analysis tools and web interface into a single software package. It is fairly simple to setup and provides the programmers with a simple interface to populate a database with log data. In order to populate the database, Logstash provides a program that can forward data from a data-source to the Elastic Stack, which can be installed and configured directly on the data source. In case this is not possible the user can also populate the database manually by sending packages to Logstash using the network interface. Due to the intuitive web interface provided by Kibana, the data can easily be organized and browsed. Using a query language it is simple to connect and filter data. In addition, Elastic Search provides the ability to use scripts for automatic analysis of the data. All these possibilities make tools like the Elastic Stack a powerful tool for analyzing log data, keeping an overview over encountered problems and therefore managing large-scale software projects.

The generation of a report, including encountered potential problems provides the programmers with an overview over the current state of the software. The quality of this overview is always relative to the quality of the test-system.

The more components of the software are tested by the system, the higher is the confidence that the software is fully functional. Systems like the Elastic Stack can be utilized to analyze the quality of the software, without having to browse through pages of log-data. Generating a report from these data and sending it via email provides the programmers with regular feedback about the status of the software, enabling them to quickly react to newly introduced malfunctions. Revealing malfunctions as quickly as possible can save a lot of time as the memory does not need to be refreshed regarding the changes recently introduced.

The new targeting system is efficient with respect to the memory and CPU consumption as the points are pre-calculated and only have to be loaded at run-time. This newly developed targeting system resulted in a better combat behavior of the bots and will therefore be further used in bot development.

Approaching the enemy in groups is one of the behaviors resulting from the navigation target system. This is usually not a bad strategy but does not compensate for the otherwise bad performance of this system. The idea of providing the bots with paths of good players to follow, turned out to be to static for this kind of game. Also to choose an appropriate target destination, definitely more than the position of the player and a single parameter modeling the overall player distribution is necessary. However these results provided new insights on the influence of the movement behavior of the players.

8. Future Work

Kibana features a variety of plug-ins to add a variety of visualization and other useful tools[1]. Different Kibana plug-ins could be installed and evaluated to even enhance the user interface and output. Also Elastic released additional software for the Elastic Stack to process the collected data using machine learning algorithms. This software can be used to specify constraints that have to be fulfilled. The system can then be configured to send notifications if any of these constraints is not fulfilled. This approach could replace the current report process by only sending a report on demand, which is in case the system detects problems.

Currently the parser for the server only features an online-mode. This means that the server log parser can only be applied during a play-test. The log data however is available on the server for a longer period of time. Due to the fact that the data does not include any reference to specific sessions, offline parsing was skipped for the server. This however can be compensated for if the time-span for a desired playtest is known, as every log-entry features a time-stamp. Therefore also an offline mode for the server log parser could be implemented if so desired.

Ideally the automatic test system uses bots that act just like humans. This way it could be assumed that the automatic test system does not leave any problems unrevealed that could be found by the consumers. Therefore, maximizing the quality of the bots is a key task to get the most out of the automatic test system. This however is not a trivial task. Due to the complexity of the game mechanics, using scripted bots will most likely not yield a maintainable and satisfying solution. Therefore more advanced machine learning methods could be utilized to further improve the bot behavior.

[1] `https://www.elastic.co/guide/en/kibana/current/known-plugins.html`

Appendix

Appendix A.

Automatic Testing and Logging

A.1. Example Configuration File

```
{
  "Some Playtest Name" : {
    "indexName" : "desired_index_name",
    "createIndexPattern": true,
    "logSeverities" : ["ERROR"],
    "serverSetup" : {
      "name" : "local",
      "branch" : "some/SVN/branch",
      "revision" : 1234567
    },
    "botSetup" : {
      "svnPath" : "path/to/SVN/root/of/bots",
      "botRepoPath" : "path/to/bot/repository"
      "scenarionName" : "autoTest",
      "numTeams" : 2,
      "numPlayers" : 25,
      "loginName" : "@bot%team_%index",
      ...
    },
    "reportSetup" : {
      "filterFile" : "Path/to/filer/file.txt",
      "recipients" : ["somePerson@mailprovider.net"]
    }
    ...
  },
  "Some Other Playtest Name" : {...}...
}
```

Listing A.1: This is an example for a playtest configuration file. The content of this file describes the modalities for playtests and is formatted in json.

A.2. Example Reports

A.2.1. Server Report

```
1  ---------------------------------
2  Report
3  ---------------------------------
4  Playtest Settings
5    *   BOT CRASH ENCOUNTERED DURING THIS PLAYTEST!
6    *   Elasticsearch Index: nightlytest-2017.10.08
7    *   Server: [redacted]
8    *   Branch: https://[redacted]
9    *   Revision: 866542
10   *   Space ID: 983136
11
12 ---------------------------------
13 Message Statistics
14 Severity          Total    Unique
15 CRITICAL          0        0
16 WARNING           551      27
17 ERROR             87       5
18
19 ---------------------------------
20 Average Load
21 App                CPU      MEM
22 cellapp04          34.76    12.00
23 cellapp05          0.26     6.00
24 cellapp06          0.10     6.00
25 cellappmgr         0.10     2.00
26 cellapp01          0.15     6.00
27 cellapp02          0.05     6.00
28 baseapp01          5.13     14.00
29 baseapp02          5.16     14.00
30 baseappmgr         0.00     2.00
31 ...
32
33 ---------------------------------
34 Detailed Messages
35 CRITICAL
36 Total#  UID      Message
37
38 WARNING
39 Total#  UID      Message
40 116     4151     BaseApp (base/account_helpers/doClientCmd.py, %d): DEVMODE: CLIENT ...
41 116     4153     BaseApp (base/account_helpers/doClientCmd.py, %d): DEVMODE: CLIENT ...
42 58      4165     CellApp Entity::pySetAttribute(%d): Avatar._Avatar__visibilityHandl...
43 ...
44
45 ERROR
46 Total#  UID      Message
47 58      4154     BaseApp (base/client_commands.py, %d): (%s, %s) (%s, %d, %d, %d, %s)
48 24      4168     CellApp (base/scripts/common/GarbageCollectionDebug.py, %d) : gcWri...
49 2       4177     CellApp (server_common/database/_connections.py, %d):Traceback (most...
50 ...
51
52 ---------------------------------
53 Filtered Messages
54 Total#  UID      Message
55 1       4176     CellApp (cell/DirectVisibility.py, %d): DirectVisibility.__onExtra...
56 ...
```

Listing A.2: Example of the plain-text version of a server report created by the reporter. It summarizes the most important information about the server logs of a playtest to provide the server programmers with a quick overview. The report is sent to the programmers via email.

A.2.2. Client Report

```
1  --------------------------------
2  Report
3  --------------------------------
4  Playtest Settings
5
6    *   Elasticsearch Index: nightlytest-2017.10.08
7    *   Server: [redacted]
8    *   Branch: https://[redacted]
9    *   Revision: 866542
10
11 --------------------------------
12 Message Statistics
13 Severity        Total   Unique
14 WARNING 15683   116
15 ERROR   2561    1057
16
17 --------------------------------
18 Detailed Messages
19 CRITICAL
20 Total#  UID     Message
21
22 WARNING
23 Total#  UID     Message
24 1325    298     [Connect] SimpleClientEntity::methodEvent: Call of unimplemented ...
25 850     306     [Connect] SimpleClientEntity::methodEvent: Call of unimplemented ...
26 264     302     [Connect] SimpleClientEntity::methodEvent: Call of unimplemented ...
27 ...
28
29 ERROR
30 Total#  UID     Message
31 29      1341    [Network] %d reports of %s on %s in the last %dms
32 ...
33 1       1322    [Network] Packet::validateChecksum: Packet from ( %d.%d.%d.%d:%d )...
34 ...
35
36 --------------------------------
37 Filtered Messages
38 Total#  UID     Message
39 1       199     [ResMgr] XMLSection::asLong: Invalid value %s in section %s
40 ...
```

Listing A.3: Example of the plain-text version of a client report created by the reporter. It summarizes the most important information about the client logs of a playtest to provide the programmers with a quick overview. The report is sent to the programmers via email.

Appendix B.

Bots

B.1. Example Weakpoint File

```
1;1.29135026872;0.363033015172;0.327901079817
1;-0.00755119209107;0.324493544279;-2.69284099013
1;-1.28719511819;0.391175521457;0.40204937737
1;0.0114201228843;0.268100619781;2.28716072878
2;0.639223727205;0.427212675886;0.857217384827
2;-0.66058504592;0.341318574154;0.815299413536
2;0.883083464961;0.378630992484;-0.624302679443
2;-0.834104086117;0.356073689958;-0.776804666345
```

Listing B.1: This is the result of a weak-point extraction. The extractor produces a CSV-file that can be imported by the bots at run-time. This file contains a weak-point in every line. The data is delimited by semicolons. The first entry denotes the avatar part, the rest of the entries form the relative transformation of the weak-point in relation to the root of this part.

B.2. Example Trajectory File

```
1  id;t_fl;t_p;f1_x;f1_y;f2_x;f2_y;f3_x;f3_y;p_x;p_y;p_z;t
2  0;0;0;-325;-166;0;-100;350;-100;-381.6;0.1;-487.6;1
3  0;0;0.1;-325;-163.4;0;-99.3;350;-100;-381.6;0.2;-487.6;1
4  ...
5  0;153.4;148.1;-325;125.2;0;96.8;350;-166;-400.3;34.9;301.5;1
6  1;153.4;148.2;-325;125.1;0;97.1;350;-166;-143.1;4.9;-392;1
7  1;153.4;148.2;-325;125;0;97.4;350;-166;-143.1;4.9;-392;1
8  ...
9  1;265.6;267.3;-325;166;0;119.5;350;-100;-245.8;31.6;226.7;1
10 2;265.6;267.4;-325;166;0;119.6;350;-100;271.6;11.1;-185;1
11 2;265.6;267.4;-325;166;0;119.8;350;-100;271.6;11.1;-185;1
12 ...
```

Listing B.2: This is the result of a trajectory extraction. The extractor produces a CSV-file that can be imported by the bots at run-time.

Listing B.2 depicts parts of the resulting csv-file after a trajectory extraction. The entries, delimited by semicolons, denote:

1. The trajectory id **(id)**,
2. the non-interpolated time-stamp of the front-line **(t_fl)**,
3. the time-stamp of the player **(t_p)**,
4. the x and y coordinates of the front-lines of the three different lanes **(f#_x / f#_y)**,
5. the players position **(p_x / p_y / p_z)**
6. and the players team **(t)**

Bibliography

[1] Chutima Boonthum-Denecke et al. *Cross-Disciplinary Advances in Applied Natural Language Processing: Issues and Approaches*. IGI Global, 2011 (cit. on p. 12).

[2] Dorothy Graham et al. *Foundations of Software Testing: ISTQB Certification*. Cengage Learning EMEA, 2008 (cit. on p. 9).

[3] Henry Linger et al. *Constructing the Infrastructure for the Knowledge Economy: Methods and Tools, Theory and Practice*. Springer, 2003 (cit. on p. 3).

[4] Mehryar Mohri et al. *Foundations of Machine Learning*. MIT Press, 2012 (cit. on p. 15).

[5] David Arthur and Sergei Vassilvitskii. *k-means++: The Advantages of Careful Seeding*. 2006. URL: `http://ilpubs.stanford.edu:8090/778/1/2006-13.pdf` (cit. on p. 16).

[6] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999 (cit. on p. 3).

[7] Jules J. Berman. *Principles of Big Data: Preparing, Sharing, and Analyzing Complex Information*. Elsevier, 2013 (cit. on p. 17).

[8] Christopher Bishop. *Pattern recognition and machine learning*. Springer, 2006 (cit. on p. 14).

[9] Rick David Craig and Stefan P. Jaskiel. *Systematic Software Testing*. Artech House, 2002 (cit. on p. 7).

[10] Ken Dunham and Jim Melnick. *Malicious Bots: An Inside Look into the Cyber-Criminal Underground of the Internet*. CRC Press, 2008 (cit. on p. 12).

[11] Matrtin Fowler. *Continuous Integration*. 2006. URL: http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf (cit. on pp. 3, 5).

[12] Robin F. Goldsmith. *Discovering Real Business Requirements for Software Project Success*. Artech House, 2004 (cit. on p. 8).

[13] Paul Hamill. *Unit Test Frameworks: Tools for High-Quality Software Development*. O'Reilly Media, Inc., 2004 (cit. on pp. 7, 8).

[14] Anne Mette Jonassen Hass. *Guide to Advanced Software Testing*. Artech House, 2008 (cit. on p. 7).

[15] Linda G. Hayes. *The Automated Testing Handbook*. Software Testing Institute, 2004 (cit. on p. 11).

[16] Jez Humble and David Farley. *Continuous Delivery*. Addison-Wesley, 2015 (cit. on pp. 1, 3–5, 9).

[17] Hubertus B. Keller and Erhard Plödereder. *Reliable Software Technologies Ada-Europe 2000: 5th Ada-Europe International Conference Potsdam, Germany, June 26-30, 2000, Proceedings*. Springer, 2006 (cit. on p. 9).

[18] Dinesh Maidasani. *Software Testing*. Firewall Media, 2007 (cit. on p. 8).

[19] Brian Marick. *Testing Quadrant Diagram*. 2003. URL: http://www.exampler.com/old-blog/2003/08/21.1.html#agile-testing-project-1 (cit. on p. 10).

[20] Tom M. Mitchell. *The Discipline of Machine Learning*. 2006. URL: http://www.cs.cmu.edu/~tom/pubs/MachineLearning.pdf (cit. on p. 14).

[21] Stephen Nelson-Smith. *Test-Driven Infrastructure with Chef: Bring Behavior-Driven Development to Infrastructure as Code*. O'Reilly Media, Inc., 2013 (cit. on p. 10).

[22] Nikhil Pathania. *A beginner's guide to implementing Continuous Integration and Continuous Delivery using Jenkins*. Packt Publishing Ltd, 2016 (cit. on p. 3).

[23] Robert H. Spencer and Randolph P. Johnston. *Technology Best Practices*. John Wiley & Sons, 2003 (cit. on p. 3).

[24] Richard S. Sutton. *Reinforcement Learning*. Springer, 1992 (cit. on p. 15).

[25] Koray Kavukcuoglu et al. Volodymyr Mnih. *Playing Atari with Deep Reinforcement Learning*. 2013. URL: https://arxiv.org/pdf/1312.5602v1.pdf (cit. on p. 12).

[26] James A. Whittaker. *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Pearson Education, 2009 (cit. on p. 10).

[27] Junjie Wu. *Advances in K-means Clustering: A Data Mining Thinking*. Springer, 2012 (cit. on p. 16).