HERIOT-WATT UNIVERSITY

FINAL YEAR DISSERTATION

Improving the Security and Correctness of a Historical Massively Multiplayer Online Role-Playing Game Server

Author:

Helen RANKIN

H00124425

Supervisor:

Dr Hans-Wolfgang LOIDL

Second Reader:

Prof. Oliver LEMON

Submitted as part of the requirements for the MEng in Software Engineering

in the

Department of Mathematics and Computer Science

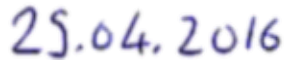
April 25, 2016

Declaration of Authorship

I, Helen RANKIN, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: Helen Rankin

Date: 25.04.2016



HERIOT-WATT UNIVERSITY

Abstract

Department of Mathematics and Computer Science

MEng in Software Engineering

Improving the Security and Correctness of a Historical Massively Multiplayer Online Role-Playing Game Server

by Helen RANKIN

This project evaluates tools and techniques, both established and emerging, for improving the security and correctness of distributed client-server applications. As a case study, this project examines the JominiEngine, an engine for massively multiplayer online games (MMOs), which can be viewed as a stress test for distributed and multi-threaded systems due to the high load and security issues that are endemic to MMOs. Static and dynamic analysis are evaluated through selected tools and libraries, as well as emerging techniques from distributed language design, and their suitability for large client-server applications is reflected on. Specifically, this project evaluates Microsoft Code Contracts, Microsoft Unit Test framework and dynamically checked session types in terms of error detection and prevention for distributed systems. In addition, this project investigates the impact increased security and session-based communication has on code complexity, runtime and memory consumption. While the immediate goal is to improve the security and correctness of the JominiEngine, the results are more broadly applicable to large multi-user distributed systems and massively multiplayer games more specifically. To conclude, this project recommends a hybrid (static and dynamic verification) approach to Test Driven Design and debugging, as well as (potentially) statically-checked session types for robust communications.

Acknowledgements

My thanks go to my supervisor, for his advice and tireless support for this project, without which this project would never have been completed. I also extend my gratitude to Calvin Houliston for his patience and motivation, and to my family for supporting my studies for all these years and encouraging me to aim high.

Dedicated in memory of Vindhya.

Contents

List of Figures

List of Abbreviations

MMO	**M**assively **M**ultiplayer **O**nline (games)
RPG	**R**ole **P**laying **G**ame
MITM	**M**an **I**n **T**he **M**iddle
ABCD (project)	**A** **B**asis for **C**oncurrency and **D**istribution
TDD	**T**est **D**riven **D**evelopment
IL	**I**ntermediate **L**anguage
EAL	**E**valuation **A**ssurance **L**evel

Chapter 1

Introduction

This chapter describes the objectives of this project, as well as the problems this project aims to solve. Also included is a short description of the emerging technique from distributed language design, session types, and a description of what is covered in the project, how the project will be evaluated, the adaptations that have been made since the initial project design, and legal, social, ethical and professional issues relevant to this project.

1.1 Project Aims & Objectives

The aim of this project is to use a combination of established and emerging software tools and techniques to improve the security of the JominiEngine described in section 1.4.1, and to evaluate the effectiveness and applicability of these techniques to large-scale distributed systems. In particular this project aims to use the communication protocol description language "Scribble" (see section 2.3.1) to model the communication protocol; to redesign the communication protocol from stateless to session-based with dynamically checked session types (described in section 2.1); to determine the impact this has on performance; and to use and compare established techniques for verifying the correctness of the server by recording the range of bugs detected.

Due to a lack of statically-checked session type implementations in C# (the language the case study is written in) the focus will not be on implementing true session types in the JominiEngine, but on modelling the communication using session types, validating these models

and refactoring the game engine to use an off-line approach to session types as the communication protocol. While the integrity of this implementation cannot be guaranteed by the type system, it can be compared against valid models to ensure it adheres to the allowed message sequences. This allows the effectiveness of modelling communication on session types to be examined without attempting to re-write entire session type libraries into C#.

The primary objectives of this project can be divided into the following:

- To investigate the impact improvements to confidentiality and integrity of communications has on time and memory performance
- To make use of current research into session types to improve the security and readability of the communication protocol
- To evaluate session types as a means of ensuring correctness of communication, i.e. that the order and type of messages is consistent with the communication protocol. To ensure a valid flow of messages, an offline approach to session types shall be used
- To implement a means of enabling the client and server to authenticate each other, and to determine the impact this has on time and memory performance
- To create a series of unit tests to dynamically check whether the functions under test perform correctly, and to gather the results
- To add Code Contracts annotations to several functions and classes, and record the result of running the static analyser on them
- To improve correctness of the server-side code using the results above, ensuring the code performs the intended function and that code integral to the security and functioning of the JominiEngine has been thoroughly tested and verified
- To create a test suite that runs through a standard flow of messages between client and server, and to measure how the time and memory consumption of running this test suite is affected by both sessions and encryption

- To compare static and dynamic analysis in terms of the number and range of bugs detected, as well as time and effort required to implement and relative ease of implementation

Of these, the focus shall be on the security of the communication through the communication protocol. In particular this project examines session types, an active research area with few implementations. Modelling the communication using session types while implementing an offline version of them in C# may contribute new findings to this field.

Connection security and client authentication is a key feature of any server. This is a well researched area with standard security practices, such as certificate-based authentication and secure key exchange protocols, and this project uses these established methods to authenticate clients and provide secure connections to the game server. The effect this had on the time and memory required to authenticate and communicate with clients has been measured, and Chapter 4 asserts that the minor impact on time is offset by the benefit of increased security. As there is less research into commercial coding tools and session types, as well as their impact on large networked projects, the focus is more on this rather than on authentication and connection security.

Aside from improving the JominiEngine, the aim is to discover *how* best to do so. Having selected several techniques and tools to aid development and improve correctness, the aim is to determine which benefit the project while not impacting performance or requiring a great deal of learning. To evaluate this, a test run has been created that executes a sequence of actions on the server and measure the memory used and time taken. The number and type of bugs found during testing was also recorded to determine the effectiveness of each tool and technique in detecting or preventing bugs. The difficulty of implementation is purely subjective and is covered in Chapter 4.6. The findings and conclusions are summarised in Chapter 5, and recommend session types and Test Driven Development (TDD) with added static analysis.

1.1.1 A Brief Note on Session Types

Session types are being pursued as a means of using type systems to increase the reliability and correctness of distributed systems. The core concept is that message type and order can

be statically checked much like any other type in high-level languages, resulting in robust communications while also decreasing development time and cost.

While much has been accomplished in realising session types, they have not yet been implemented for C# and remain far from becoming a mainstream language feature. As such, this project implements a dynamic approach to session types. More can be read on session types in section 2.1.

1.2 Problems Solved

This project answers the question of how security and correctness can be enforced across a large distributed system which has had limited or no structured testing. In particular, it focusses on what techniques are most appropriate for the domain of massively multiplayer online games (MMOs), including the complexity of each technique, the time required, and the impact on memory and performance.

As session types are an emerging language feature, and static analysis has largely been confined to systems that require high reliability and integrity (such as safety-critical systems), there is limited existing information about how these may be applied to large distributed and concurrent systems. In Chapter 4 the improvements to the correctness of the client-server communication with respect to the communication protocol are described, and Chapter 5 concludes that session types and static analysis are appropriate for large distributed systems (including MMOs), and outlines how they may be used and what benefit they provide.

1.3 Project Evaluation

While the focus of this project is on improving security and investigating techniques and tools for improving code reliability, it is important to remember that the case study is a MMO role-playing game (MMORPG) server which will need to cope with large numbers of connections and a high strain on resources, and so the performance of the JominiEngine should be compared to determine what impact the additional security has. Additionally, the maintainability

of the code should be taken into account to assess whether changes to the code have made it more or less manageable. The evaluation of this is described in section 1.3.3.

1.3.1 Static versus Dynamic analysis

To evaluate the effectiveness of dynamic analysis, a series of unit tests has been created using the Microsoft Unit Test framework (Chapter 2.3.4) to determine whether the replies given by the server and the state of the game are consistent with the messages sent by the client. The methods tested were logging in, maintaining an army, recruiting troops, travelling, adjusting fief expenditure, spying and attacking. Logging in was chosen as it is a core security requirement, while the others were chosen to reflect a wide range of actions available to the client, as well as being some of the more complicated actions to carry out.

The number and details of bugs encountered during the running of these tests was recorded and categorised, which allowed static analysis and dynamic analysis to be evaluated in terms of range of bugs found.

The evaluation of static analysis was carried out using Microsoft Code Contracts (Chapter 2.3.3). By setting the static analyser to search for potential sources of error (such as NullReferenceExceptions) and adding preconditions and postconditions to several functions, the number and range of issues detected by the analyser could be recorded and analysed.

In the results section (see Chapter 4) the type of sources of error (both real and potential) detected by static and dynamic analysis has been compared. This indicates what type of error (from uncaught exceptions to incorrect behaviour) each technique is best suited to finding. Results indicate that static analysis is best suited to detecting potential common exceptions (such as numbers being out of range), whereas dynamic tests are more suited to logic errors and issues involving dynamic collections.

1.3.2 Performance

To determine whether the additional security measures and changes from a stateless to session-based communication protocol had an impact on the time and memory performance, a small test run was created that ran through a series of actions comparable to normal game-play,

timed how long each stage took and measured the maximum memory consumed. This was run 10 times each for four different versions of the program:

- Stateless communication protocol with unencrypted messages
- Stateless communication protocol with encrypted messages
- Session-based communication protocol with unencrypted messages
- Session-based communication protocol with encrypted messages

The test began with a client logging in, followed by recruiting troops, travelling to another fief, and spying on the fief they had travelled to. Times were measured from client sending the message to the moment the expected reply was received, and the memory usage was measured after each action. This tested the time for a wide range of actions, and reflected how long a client could expect to wait (with a good connection) to receive a reply.

1.3.3 Maintainability

The code maintainability was measured using the in-built code metrics tool for Visual Studio (see section 2.3.2), which calculates a value for each class and method based on the number of different paths of execution (cyclomatic complexity), the depth of inheritance, how dependent the code is on other classes (coupling) and the lines of code in Intermediate Language (IL). This final maintainability value is between one to one hundred, with higher values indicating better maintainability, and is calculated using the following formula (Microsoft Developer Network, 2007):

$$\begin{aligned} x = &(171 - 5.2 * log(\text{Halstead Volume}) \\ &- 0.23 * (\text{Cyclomatic Complexity}) \\ &- 16.2 * log(\text{Lines of Code}) \end{aligned}$$

$$\text{Maintainability Index} = MAX(0, x * 100/171)$$

While the validity of this measurement is dubious, as pointed out in a blog by software engineering professor Arie van Deursen (Deursen, 2014), it is sufficient for this project as the analysis is not concerned with what is classified as a "good" maintainability rating (as this is highly subjective), but rather what impact changes to the project have had on maintainability.

1.4 Scope & Context

Security and correctness are challenging in distributed systems, as both the computation and coordination are potential sources of error. MMOs present additional challenges in that they are intended to be played by many people simultaneously, requiring massive coordination and putting strain on resources. Additionally, a proportion of MMO players are experienced in modifying and hacking games, and are skilled in identifying and using security flaws for their own gain. This makes the JominiEngine a good stress test for both established and emerging techniques for improving security.

1.4.1 Project History

The JominiEngine (Bond, 2015) is a game engine for online massively-multiplayer role-playing games (MMORPGs) designed to be used by thousands of players in a large-scale, distributed setting. The core game engine was developed in C# as a monolithic application in 2015 by Heriot-Watt University master's student Dave Bond, and features land, army and family management in an accurate historical setting. Designed to evaluate technologies for building and designing game servers, the key considerations were scalability and modularity, which are reflected in the choice of database, Riak (Klophaus, 2010), and the structure of the code. While the core game functionality was implemented, due to time restraints the project was implemented as a monolithic structure rather than client-server structure, and was scaled back to exclude the multiplayer and networking functionality. Figure 1.1 illustrates the key actions that can be taken by the player.

As part of a larger effort in designing and implementing a game engine for Serious Games in history education, I was employed to extend the JominiEngine as part of an internship at

Heriot-Watt university. During this I designed and implemented the networking functionality, refactored the existing code to support multiple players, built a game client in the game development platform Unity, and added extra functionality to the core game engine by adding a "dirty deeds" system which enabled kidnapping, spying and ransoming.

For the work in the final year project described here, the focus shifted to improving the security and correctness of the JominiEngine. Security and stability across all aspects should be taken into account in any software project, and this particularly challenging for MMORPGs which involve thousands of skilled players, many with the motivation to try and subvert security measures. Flaws in the code or in the communication layer can take the server offline or have unintended consequences for players, both of which can result in decreased user satisfaction and the exposure of private data. Another problem common to large-scale, distributed systems is that the sheer size of the code makes securing and testing code challenging. This project aims to improve the security of the JominiEngine, and in doing so evaluate various methods of improving server-side security and make recommendations for the design and implementation of secure distributed systems.

1.4.2 Project Scope

As a final year project, this is a solo project running from September 2015 to mid-April 2016, but is built on a previous implementation of the game engine as described in section 1.4.1. The JominiEngine consisted of over 30,000 lines of code across 35 classes before changes implemented in this project, and implements a range of features (such as land management and combat), a No-SQL Riak database back-end, several administrative functions and a client-server architecture suitable for distributed systems. Due to the large existing code size and complexity, changing or refactoring code can be time consuming and may produce unintended results.

For the initial implementation the focus was on functionality, whereas for this project the focus is to use a range of techniques to improve security and correctness. Rather than attempting to apply these techniques to all parts of the engine (due to the many months this would take), different techniques have been applied to select parts of the engine and assessed. This project selects representative tools from different classes of established and emerging tools:

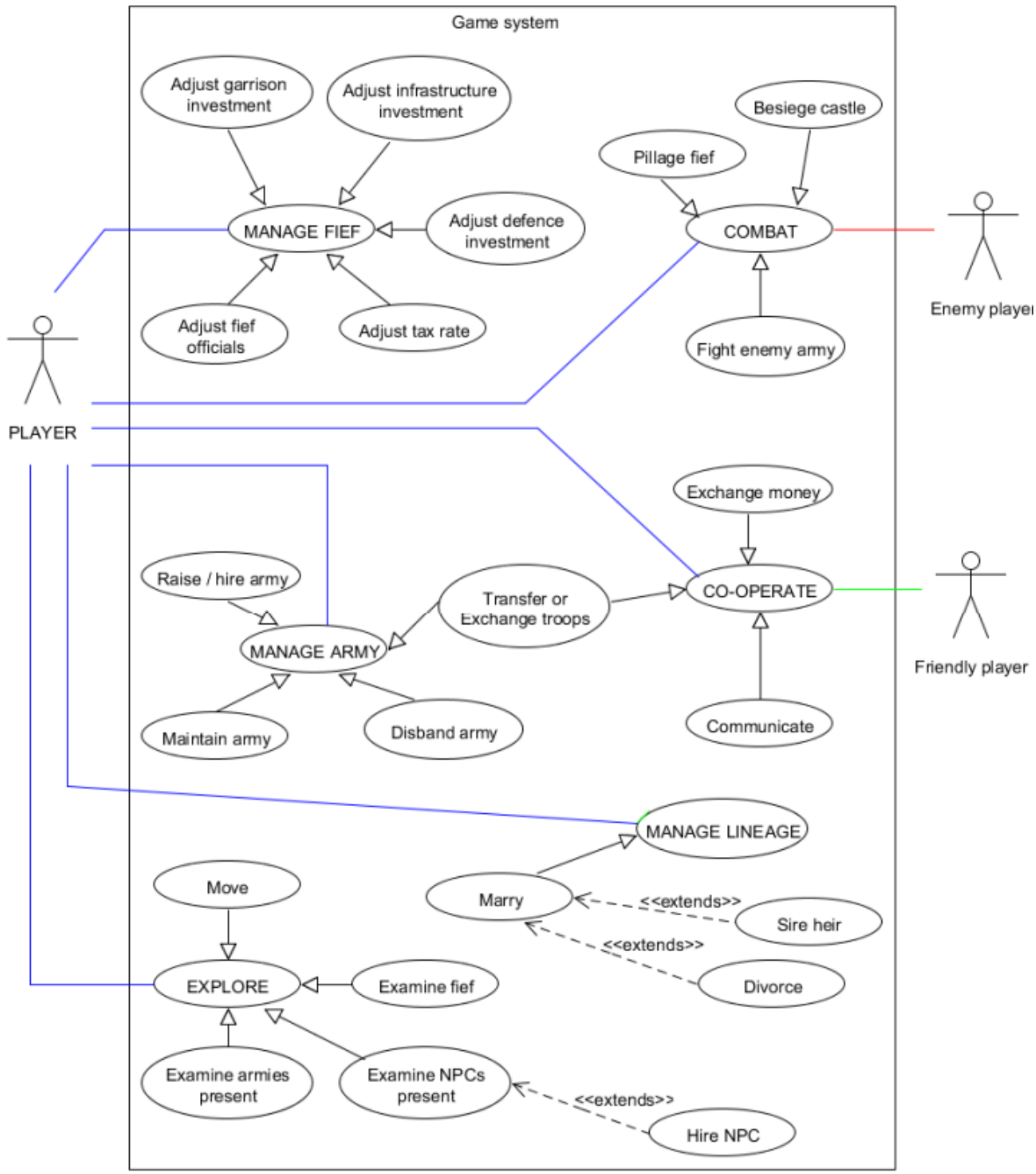


FIGURE 1.1: JominiEngine UML diagram, taken from Dave Bond's thesis (Bond, 2015)

Microsoft Code Contracts for static analysis, Microsoft Unit Test framework for systematic runtime checking, and session types for a secure communication protocol.

This project also examines the applicability of session types to the domain of large, distributed client-server systems. Session typing aims to use static type checking systems to guarantee correctness and deadlock-freedom in communication protocols, and can be read about in more detail in Chapter 2.1. While session types are an active area of research there is currently no existing implementation of static type-checking for sessions for the full C# language. Therefore, this project takes a dynamic approach of checking properties that can be encoded using session types, and will talk extensively in Chapter 5 about the potential benefits session-types can bring to large distributed systems such as the JominiEngine, as well as making recommendations for the development of distributed systems in general.

The scope of the project is summarised below:

- The benefits of static analysis shall be explored through the use of Microsoft Code Contracts, based on pre and post-condition verification, as an example of classic sequential techniques for secure coding
- The benefits of dynamic analysis shall be explored through the use of dynamic (runtime) whitebox unit tests- with assertions to check the messages received and game state is as expected - created using Microsoft Unit Test framework
- The tests mentioned above shall be created for functionality which is either key to the security of the server, non-trivial or has been changed significantly over the course of the project
- The benefits of using state-of-the-art code analysis, refactoring and generation tools shall be explored using the ReSharper extension for Visual Studio 2013
- The benefits of using the concept of session types for dynamic checks of protocol coherence, as well as the effect on time and memory usage of moving from a stateless to a session-based system, will be explored and a comparison made between the implementation and a true session-typed communication protocol

- The code complexity and maintainability shall be measured using Visual Studio 2015's Code Analysis features

1.4.3 Security

This section is based in content and structure on the Common Criteria for Information Technology Security Evaluation (CC), a framework for specifying the security functional and assurance requirements of a system through a protection profile. CC is used to provide some assurance that the security requirements of a product have been carefully defined, analysed, implemented and tested, and while this project will not be evaluated by an external party, the CC structure still provides a clear and thorough way to define security requirements. The CC is described in more detail in the official publication (Common Criteria Maintenance Board, 2012).

Protection Profile

The JominiEngine is an online multiplayer game engine, and faces similar security challenges to other games in this genre. Clients will connect across the internet to a single machine running the game and will send sequences of messages across indicating what actions they wish to take. While a message makes sense as part of one sequence, it might have to be prohibited in other sequences; as such message sequences and message types must be checked to ensure they adhere to the communication protocol.

Each client has one PlayerCharacter through which they can manage their land, armies and household, and should only be able to perform actions relating to their own PlayerCharacter. At the moment the game is free to play, and no registration or sign-up process exists.

Clients ought to be able to connect from any machine with minimal set-up, requiring only a valid game client software.

Security Target

During this project, the communication protocol between client and server has been changed from stateless to session-based. This brings about new security concerns, as certain sequences

of messages should not be permitted (for example, a Log In request is not permitted after successfully logging in, and requests to perform an action in-game are forbidden until logging in). Additionally, the way several messages are handled has changed significantly during the project, and now use complicated asynchronous tasks to retrieve and process replies from the client.

As clients will be connecting to the server over the internet, there is the possibility that attackers wishing to obtain information from them could masquerade as the server and fool the client into providing personal details. Ensuring clients are able to identify and authenticate the server is a key security concern.

Password security is an obvious security need, due to the prevalent use (and re-use) of passwords in a vast range of services. As such, keeping passwords secure both in storage and during transmission is a key security concern.

Although sensitive information will never be transmitted as part of normal gameplay, as a competitive real-time strategy game players may attempt to sabotage each other by intercepting messages and either changing them to invoke unintended actions for their characters, or reading them to obtain information. While this would be a substantial amount of effort for very little gain, it would be beneficial to take some action to provide some measure of message integrity and confidentiality. Messages do not need to be undecipherable; increasing the time taken to decode and re-encode a message by even a minute will render most attacks ineffective (due to the large number of messages passed and the relative insignificance of a single packet) and will hamper attempts to sabotage a game in this manner. This is a secondary security concern, as it affects player experience but does not place clients' information at risk.

Security of the Administrative Components

To ensure password security, the password should be verified by using a salted hash of the password, rather than the password itself. Chapter 3.4.2 describes the method used to verify passwords, which is an established technique for password security that ensures not only that the password is never transmitted, but that the password hash (which could be used to log in) is kept secure.

Client-Server Security

The client must be able to authenticate the server in order to prevent attackers from masquerading as a trusted server. This will be achieved through X509 certificate verification, as it is a widely used and well-recognised technique for authentication. The server's certificate (or the certificate of the root authority which signed the certificate) should be stored on the client machine during installation of the game client, enabling the client to validate the server's certificate during the connection process.

Messages sent between the client and server should be encrypted in order to provide some secrecy and to discourage attackers who may wish to read or alter them. This encryption should prioritise speed over effectiveness, as attempting to encrypt large volumes of messages has the potential to slow down the server and cause latency issues for clients. The AES algorithm was chosen for this purpose, as it is a widely used encryption standard with high speed and low memory overheads. However, as a symmetric algorithm it requires a shared key to perform encryption and decryption, which must be transmitted securely.

One common form of attack against distributed systems is the "Man in the Middle" attack (MITM), in which an attacker intercepts the connection between the client and server and impersonates the other party. This is a known vulnerability of certificate-based communication; however, it can be protected against by using signed certificates. The client sends plain data to the server, and the server signs this with its private key. The client can compare the signed data to the original data and certificate, and can determine its authenticity. While a MITM attack could intercept the connection, it would fail the signature test and the client be aware that the connection is insecure.

Security Assurance Requirements

As the log in functionality is the main potential source of security concern, it should be extensively tested. Additionally, both valid and invalid sequences of messages should be tested to ensure the communication protocol handles sequences of messages as expected.

Extensive testing should also be performed on functionality which has been changed to use asynchronous tasks to get a client's response.

Evaluation Assurance Level

As there is a fairly limited personal risk to those playing the game, and the game itself is intended to be used largely for education and entertainment purposes, the Evaluation Assurance Level (EAL) has been set at level 1: only functional testing (or black box testing) is required. Functional testing should be applied as per the Security Assurance Requirements section.

However, it should be noted that games are popular targets for skilled attackers, and therefore is a good case study for the security of distributed systems. As this project investigates techniques for improving the security and correctness of distributed systems, both functional testing and static analysis has been carried out, despite this being above the normal requirements of the EAL.

1.5 Project Adaptations

Following the initial planning stage of the project it became apparent that the initial scope of the project was too broad (particularly in the range of techniques and tools to use) and in order to create one detailed and coherent document in the time given it would be necessary to reduce the project scope and scale. The focus is to use selected tools as representatives of different classes of techniques: Microsoft Code Contracts for static analysis, Microsoft Unit Test framework for dynamic analysis and session types for communication protocol verification.

During the creation of the unit tests it became clear that extensively testing every available action in-game was going to be time consuming. Additionally, there existed many similarities between how certain actions were handled, reducing the merit in testing these other actions. To save time while still obtaining useful results, it was decided to restrict testing to functionality that was integral to the security of the server, had been changed during project development or was particularly complicated.

Originally it had been intended to investigate how changes to the code affected the readability- however, readability is hard to measure and is often based on how code is commented rather than the code itself. During development it was discovered that Visual Studio's Code Analysis tools analyse code and calculates its own code maintainability rating, which can be read about in more detail in section 1.3.3. This is preferable to calculating readability, and will be measured instead.

1.6 Work Done

This section briefly summarises the work done, in order to give an impression of the amount of implementation involved. A more detailed description of the work done can be found in Chapter 3.

Sessions

Perhaps the most significant change has been from a stateless to session-based communication protocol. This involved not only a great deal of re-writing code, but also required learning asynchronous programming in C#. This change from synchronous programming to asynchronous also opened the door to several consistency issues, or "race conditions", which came about as a result of multiple threads (created from asynchronous operations) interfering with each other. Fixing the worst of these required a great deal of design and planning involving EventHandlers, thread wakeups and locking. This is covered in more detail in Chapter 3.1.

Refactoring

To make the transition from stateless to session-based possible, it was necessary to refactor a large part of the main game controller. This controller is responsible for translating a client's request into a response, and was previously implemented entirely in a case-switch. The case switch was previously 1332 lines of Intermediate Language (IL) code (or over 2500 lines of C# code) for validating and processing 60 different types of client request. The sheer size alone made it infeasible to turn this into a session-based implementation, and the need

for asynchronous execution further complicated matters. During refactoring each case in the case-switch was implemented in its own method, which could be either synchronous or asynchronous, which left only the logic for extracting the data from the client's request in the case-switch. While the task was time-consuming, it had the effect of dramatically increasing the maintainability, as shown in Chapter 4.5, where the maintainability increased from 0 to 11 in the stateless implementation (and 10 for the session-based implementation).

In addition, the process of validating incoming data was previously done per-request-type, resulting in a great deal of repeating code. This was replaced with a single, consistent means of validating data, which can be read about in more detail in Chapter 3.7. Refactoring this was time consuming, but reduced code repetition greatly.

Security

Prior to this project only password security was implemented; there was no way for the client to authenticate the server, and no means of encrypting messages. During this project a certificate-based authentication process has been implemented, which includes a means for a certificate to be sent to the client, signed messages for server authentication, a key-exchange process to securely share a symmetric encryption key between client and server, and for messages sent between client and server to be encrypted with this shared key.

Test Client

To test the server it was necessary to build a basic client that would be capable of connecting to the server and sending requests. This involved configuring a client to connect and send all required log in information, as well as creating all the requests that would be used during testing. Though not a main part of the project it was integral to the testing, and was time consuming to make.

Unit Tests

To perform dynamic analysis on the JominiEngine, a series of unit tests were created to test several client-requests with a wide range of parameters as well as for invalid sequences of

messages. 51 tests were created for both the session-based and stateless implementations, and each of these tests had to be run after making and adjustments to the code. In addition, if any tests failed (or produced otherwise unexpected results) the problem had to be identified, logged and fixed.

Versions

As this project involved testing both session-based and stateless communication protocols, it was necessary to maintain two versions of the JominiEngine (while the results refer to testing four versions, the difference between the encrypted and unencrypted versions was simply whether the client supplied an encryption key or not, and did not require additional maintenance). All issues found in one implementation had to be investigated in the other; and any changes to non-communication based components had to be merged into both implementations. While this was made easier through Git version control (allowing easy switching and merging of versions), it still required effort to maintain both versions.

1.7 Relevant Professional, Ethical, Legal and Social Issues

This project builds on existing research across multiple disciplines and sources, and credits these in the References section. While the work done in this project is my own, the original creation of the JominiEngine is the sole work of Dave Bond (Bond, 2015), and I am in no way involved in the creation and development of the tools and techniques mentioned in this project.

While this project describes some forms of attacks on distributed systems, these attacks are well known and this project does not encourage or provide instructions for carrying out these attacks. Rather, this project outlines some methods of reducing their effectiveness.

Microsoft Visual Studio Community 2013 is a licensed piece of software, and is available free for individuals working on their own projects. This project does not alter or re-distribute any of Visual Studio Community 2013's code and is within full use of the Microsoft software licensing terms.

JetBrains ReSharper Ultimate is a commercial piece of software which is available for free to students. This project uses ReSharper Ultimate under the student license, and does not attempt to modify, distribute or reverse-engineer any of ReSharper's code or files. As this project is purely academic, the use of ReSharper for this project is within the JetBrains student license agreement.

This project is assessed work for Heriot-Watt University, and may be visible to those outwith the university. This project aims to meet the high standard of work set by Heriot-Watt University, and to contribute new research to the field of security and correctness for distributed and concurrent systems.

The use of this project should be credited and referenced appropriately.

Chapter 2

Background

2.1 Session Types

Session types are being researched and developed as a means of using static type checking systems to verify communication protocols in distributed and concurrent systems. This verification determines that the right messages are being sent in the right order, and that there is no risk of any communicators either waiting endlessly for other communicators (deadlock) or getting trapped in a cycle of execution (livelock). Whereas conventional deadlock prevention techniques involve timestamps or resource locks, the goal of session types is to build on static type-checking systems to validate protocols without the need for locking or dynamic checks. Several researchers (such as those involved in the "A Basis for Concurrency and Distribution" (ABCD) project (Wadler, 2016)) are working on making session types a reality.

To model communication protocols, the Scribble language and protocol (Yoshida et al., 2013) has become a commonly-used tool among ABCD researchers for describing application-level communication protocols. Put simply, it abstracts a rather complicated web choreography description using message signatures, which made it simpler to build and use. The Scribble project features tools to model, validate and run simulations on session-typed protocols and is the first of its kind, representing a step towards an end-point implementation of session types. While the majority of session type theory involves detailed knowledge of the π-calculus and/or linear logic, Scribble is comparatively simple to understand and the tools are straightforward to install and use. This makes it useful for this project, as the fairly novel and complicated concept of session types can be illustrated clearly using Scribble protocols. For

example, the log-in process used in the JominiEngine involves several authentication stages between client and server, during which either party can disconnect if they feel the other has failed the authentication checks. While complicated in practice, this can be clearly represented in Scribble as in listing 2.1. Scribble can also be used to project protocols, so that the protocol

```
1
2  global protocol LogIn(role Client, role Server) {
3          choice at Server {
4                  SendSaltAndCert(ProtoLogIn) from Server to Client;
5                  choice at Client {
6                          SendHashAndKey(ProtoLogIn) from Client to
                               Server;
7                          choice at Server {
8                                  SendClientDetails(ProtoClient) from
                                       Server to Client;
9                          }
10                         or {
11                                 Disconnect(String) from Server to
                                       Client;
12                         }
13                 }
14                 or {
15                         Disconnect(String) from Client to Server;
16                 }
17         }
18         or {
19                 Deny(String) from Server to Client;
20         }
21 }
```

FIGURE 2.1: The Scribble protocol for LogIn, from a global perspective

can be viewed from the perspective of one of the communicators. Listing 2.2 shows the LogIn protocol from the Server's perspective. Scribble protocols can be visualised as finite-state machines, which may be recognisable to those not familiar with communication protocols. Figure 2.3 shows an overview of the JominiEngine communication protocol as a state machine from the server's prospective, including the LogIn protocol. At the time of writing, session types remain a niche, yet active, research area and are not used in mainstream software development. However, session types have been implemented for some mainstream languages: SessionJ (Hu

```
local protocol LogIn at Server(role Client,role Server) {
        choice at Server {
                SendSaltAndCert(ProtoLogIn) to Client;
                choice at Client {
                        SendHashAndKey(ProtoLogIn) from Client;
                        choice at Server {
                                SendClientDetails(ProtoClient) to
                                   Client;
                        } or {
                                Disconnect(String) to Client;
                        }
                } or {
                        Disconnect(String) from Client;
                }
        } or {
                Deny(String) to Client;
        }
}
```

FIGURE 2.2: Scribble LogIn protocol, showing the sequence of communication from the Server's perspective

et al., 2009), a Java-based language written using the Polyglot compiler, is one of the first to demonstrate session types being used in a popular object-oriented language. The similarities between Java and C# make this a promising advancement in relation to the JominiEngine and show that using session types to guarantee reliability and consistency in distributed systems is within reach.

In addition, a paper released as part of the ABCD project earlier this year (Hu and Yoshida, 2016) outlines a way in which session types could be implemented in existing mainstream languages using a combination of static and dynamic type checking. The approach described by Hu and Yoshida involves creating validated models in Scribble and then generating a finite state machine from the model which shows all valid transitions from one state to another. Each individual state is written as a channel type in the target language, where a channel transports messages across a boundary between two or more peers. The incoming and outgoing messages in each channel are specified by the protocol, and are therefore safe. To confirm that the channel is being used correctly, two runtime checks are used: the first checks that the channel instance

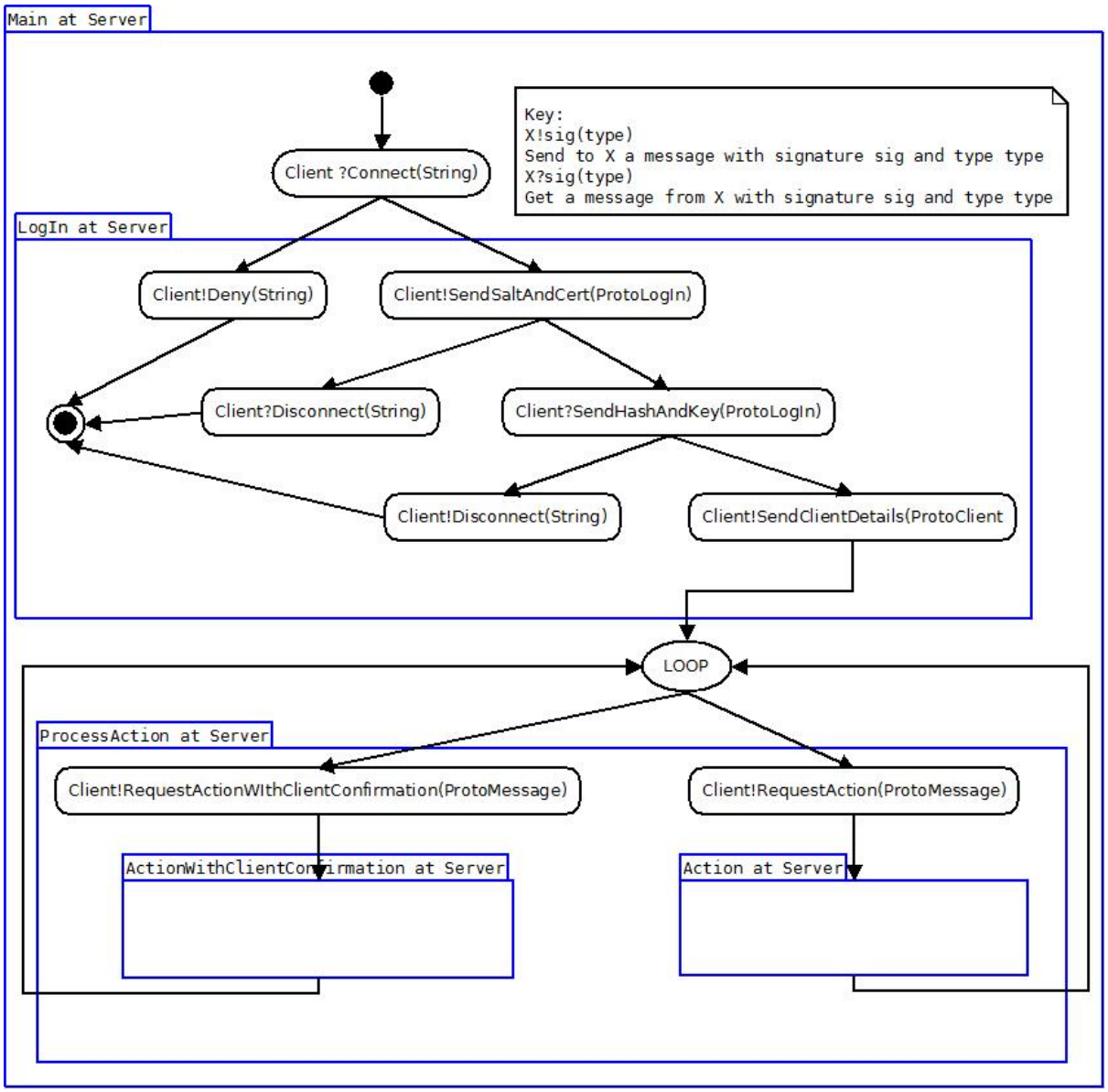


FIGURE 2.3: Diagram showing the communication protocol from the server's perspective a a state-machine diagram

has not been used more than once, and the second checks that it is not left unused (i.e. that it is not left inactive and expecting input). These combined result in a robust implementation of session types without requiring extensive re-writing of the language to work.

The concept of session types has existed for decades now, first being proposed by Honda

in his 1993 paper (Honda, 1993). Honda imagined concurrency as types and laid the theoretical foundation on which session types are built, combining constant types, input types and output types together in sequences and branches. This was complimented by work done by Abramsky (Abramsky, 1993) in translating linear logic into π -calculus; however, as Wadler mentions in his 2012 paper (Wadler, 2012), neither of these follow on from Curry-Howard. Curry-Howard correspondence (or Curry-Howard isomorphism) can be generalised as that programs are proofs and formulas are types for languages, which means that there is a strong link between a program and the mathematical foundation underpinning it. Without this property any direct implementations of Honda's or Abramsky's work would not have a solid link from the implementation to its mathematical theory, thereby making it harder to prove their correctness. Wadler's work builds on a variation of Abramsky's which is Curry-Howard, and proposes a language which is free from deadlocks by virtue of being derived from linear logic.

Shortly after Wadler's publication, the ABCD programme was given a grant by the Engineering and Physical Sciences Research Council (Wadler, Yoshida, and Gay, 2013). In the grant application form, session types are described as a "revolution" for distributed systems with the potential to reduce development costs, increase efficiency and improve reliability, and the project partners include major companies such as Amazon, Red Hat Labs and Cognizant Technology Solutions. Since receiving the grant the ABCD programme has achieved several important milestones in the path to a feasible implementation of session types, bringing them closer to a mainstream technique. However, distributed systems are complex and there are many challenges to realising session types.

One problem of attempting to type sessions was how timeouts should be handled. In the JominiEngine, where clients could disconnect or fail to respond at any time, ensuring freedom from errors in spite of this is necessary in order to protect the integrity of the server and its data. Bocci, Yang and Yoshida propose a solution to this issue in their conference paper on timed multi-party session types (Bocchi, Yang, and Yoshida, 2014). While still in its early implementation stages, Bocci et al's theory will have profound implications for ensuring reliability in timed or scheduled applications.

As session types continue to be researched we are seeing new ways in which mainstream

languages can incorporate them. It is likely in the forseeable future that session types will go from a niche research area to a widely recognised and used tool for developers of distributed systems.

2.1.1 Statically Checked Session Types versus a Dynamic Implementation

As previously mentioned, the lack of a session type implementation for C# makes statically checking for adherence to the communication protocol infeasible for this project. Instead, messages are verified dynamically to ensure the message contents and order are consistent with the communication protocol. In the case of the LogIn protocol, when the Client connects it is checked that the hail message contains a string. If the message is a verified username, the server sends the certificate and then anticipates the client's log in details. If the next message received is not a valid log in, the server terminates the connection; otherwise, it sends the client details and repeatedly waits for and processes the next message, stopping when the client disconnects. This can be seen in figures 3.1 and 3.2 in Chapter 3, and can be read about in more detail in the same chapter.

This implementation checks the order and type of messages received dynamically, whereas session types would use an underlying type system to statically verify message sequences. This would result in faster performance (due to a lack of checks during runtime) as well as more robust code: changes to the code may inadvertently render the JominiEngine open to invalid message sequences, whereas static types checking guarantees adherence to the protocol by the type system.

2.2 Static Analysis in Distributed Systems

The complexity of distributed systems has always presented a challenge to conventional testing techniques. As most distributed systems also make use of concurrency, analysis must check the validity of messages being received and sent, the order of messages, the safe execution of threads and the code itself.

Relying on static analysis is a tempting prospect at first glance; requiring very little additional effort on the behalf of the developer, static analysis is powerful enough to check a wide range of conditions. However, as pointed out in an article by Dr. Anderson (Anderson, 2008), static analysers are limited in several ways. Firstly, they are incapable of checking all paths of execution. Taking into account the number of loops, conditional statements, asynchronous methods and recursive calls that are common to many systems, this is understandable- there are potentially infinitely many paths of execution. However, it does present the risk that one of the paths not checked is the very one that causes the server to fall apart. Secondly, most static analysers cannot analyse libraries used in the code. Issues in how the libraries are used, or in the libraries themselves, will more likely than not slip through the static analyser.

While static analysis has its uses, the effectiveness varies; research by Goseva-Popstojanova (Goseva-Popstojanova and Perhinschi, 2015) found that when running static analysis on C++ and Java code, between 59% and 62% of known errors were detected for C++ and 67%-73% for Java. While this appears reasonable, two of the three Java analysis tools had around 25% probability of reporting a false positive. This shows that static analysis is not always a reliable technique.

Rather than relying solely on static analysers, other researchers have experimented with creating languages that better support concurrency. An example of this is P# (Deligiannis et al., 2015), which uses state-machines to avoid race conditions, and features a static analyser and a bug-finding mode. The team behind P# intended to originally use the CHESS (Musuvathi et al., 2009) tool to analyse P#; however they were able to create a more efficient implementation by implementing a depth-first search and a random scheduler that took into account the absence of race-conditions in P#. While their results are promising for avoiding race conditions, and the integration with C# would make it a potential candidate for evaluating in this project, it does not explicitly prevent deadlocks. However, as they point out in the paper, having no race conditions means that "only synchronizing operations need to be treated as visible, which greatly reduces the exploration space". This makes it possible to use conventional resource-locking techniques only when synchronizing, which simplifies the process of preventing deadlocks and could improve the efficiency of the program due to less resource-locks taking place.

While using a language specifically designed for distributed and concurrent systems may be possible for some projects, in this case due to the large amount of existing code and reliance on libraries it is not. As we have seen that pure static analysis may not be the answer, a combined approach such as the one developed by Sözer (Sözer, 2015) may be more appropriate. In his paper Sözer describes a process by which the results of static and dynamic analysis are fed into each other, the idea being that the dynamic analysis filters out false positives while the static analysis indicates potential sources of error. This is repeated, producing a report of detected errors which is typically more complete than just dynamic analysis, but with less false positives than static analysis. However, Sözer admits that the results may be affected by operating system scheduling, and as only one system was tested the results cannot be generalised.

2.3 Tools Used

This section describes the various tools employed during this project for implementing, refactoring, running and testing code, as well as for modelling the communication protocol.

2.3.1 Scribble

Scribble (Yoshida et al., 2013) is a language and tool for describing session types, which are written as communication protocols for participants. It features tools for verifying the correctness of protocols, running simulations on a protocol and "projecting" a protocol, a term used to describe rewriting a protocol from the perspectives of each of its participants. It is written in Java and can be run either from the command-line, or within the Eclipse IDE. An example global protocol written in Scribble can be viewed in listing 2.1, and the corresponding local protocol in listing 2.2. The projection feature is useful for visualising how each peer sees the communication protocol, which makes implementing the client and server side easier.

2.3.2 Microsoft Visual Studio Community 2013

Microsoft Visual Studio Community 2013 (Johnson, 2014) is a feature-rich Integrated Development Environment created by Microsoft. It can be used to develop a wide range of applications, including web services, and features built-in support for C# as well as debugging and testing tools. There are also several plug-ins that can be installed with Visual Studio to improve development, debugging and testing. One such plugin is ReSharper, which will be covered in section 4.4. Due to the wide range of features offered (as well as the support for ReSharper and Code Contracts), Visual Studio 2013 is the development environment of choice for this project.

2.3.3 Microsoft Code Contracts

Code Contracts (Fähndrich, 2010), developed by Microsoft, are a tool for Visual Studio and are part of the System.Diagnostics library for C#. With Code Contracts developers can add markings to code which specify constraints, such as preconditions, post-conditions and assertions. Code Contracts comes with both static and dynamic verification tools to detect any violations of contracts. In this project, Code Contracts will be used to detect potential bugs both statically and during run-time.

2.3.4 Microsoft Unit Testing Framework

The Microsoft Unit Testing Framework (see Chapter 11 of Johnson, 2014) is a framework for creating unit tests in Visual Studio. The framework allows methods to be marked with the ClassInitialize and ClassCleanup attributes, which indicates that these methods must be run before and after the tests, and facilitates easier setting up and finalizing of the test environment. Also supported are several types of assertions, which check that some condition is being met- this can be that the object is not null, that a boolean is true, or that an object exists in a collection.

2.3.5 JetBrains ReSharper Ultimate

ReSharper (Gasior, 2014) is a plug-in for Visual Studio which offers code analysis, refactoring and debugging tools for C# and other languages. It enables developers to enforce code

standards, analyse code quality and detect potential errors, as well as suggesting possible solutions. ReSharper also aims to help improve the readability and quality of the code, making it a valuable additional tool for the project. The ability to customise code inspections and refactor a large codebase quickly in ReSharper helped to decrease the project's development time and improved coding style.

Chapter 3

Implementation

3.1 Session Types

To create an approximation of session types without an underlying static type checking system capable of detecting violations of session type protocols, it was necessary to find ways of dynamically checking the order of messages being received and of verifying the message type. Below is outlined how the communication protocol (as shown in figures 2.2 and 2.3) was dynamically verified.

As all (valid) messages sent by a logged-in client are subclasses of the same base class (ProtoMessage), which is composed of two enums representing the action to be taken and the response code as well as several strings for including data, verifying the type of message was performed through attempting to deserialize as ProtoMessage and, if successful, checking the action request type to determine what action the client was attempting to perform. If necessary, the message would then be cast to the appropriate subtype; failure at this stage means the wrong type of message has been sent.

Message sequences were validated using the ActionControllerAsync method in the Client class. First, the Client initiates the connection with the server, sending their username as a hail message, which is handled as a ConnectionApproval type message on the server. If the username obtained from the client is unrecognised, the server denies the connection; otherwise, the server proceeds to the next verification step. This is shown in listing 3.1.

The next step verifies that the client can log in. If they cannot log in (for example, they are logged in already) the server denies the connection; otherwise, it sends the signed certificate

and salts to the client and begins the ActionControllerAsync. Finally, the protocol specifies that the next message from the client must either be a disconnect or the client's hash and key. Disconnects are handled automatically, and abort processes associated with the client. The code shown in 3.2 shows code from the ActionControllerAsync method waiting (with timeouts) for the next client message and aborting if not a log in.

This method guaranteed that the first message received was a Log In message, and instructed the LogInManager to verify the user's credentials. Next it anticipated any valid action as defined in the ActionController in the Game class- any action not recognised by the ActionController would return a MessageInvalid message. This would then repeat until the user disconnected (upon which any methods waiting on a reply would be cancelled) or timed out.

User disconnects (voluntary or through connection issues) were handled with the help of cancellation tokens in C#. When a user disconnects (or the server shuts down) the token is cancelled and the user is removed from the list of registered observers. The status of the cancellation token is checked in several places throughout the code, and upon detecting cancellation the current action is aborted. This reduces the risk of threads running beyond the lifetime of the client who spawned them, which avoids unnecessary strain on the server and reduces potential resource leaks.

The current implementation dynamically checks both message types and message sequences, but this increases runtime and does not guarantee freedom from deadlocks. In principle, the main motivation for session types is to use static type checking systems to eliminate the need for dynamic checks such as those implemented in the JominiEngine and to prevent deadlocks, thereby both increasing the efficiency and reliability of the communication protocol. Should session types come to fruition it would be beneficial to attempt to integrate them into the JominiEngine, as they are ideal for the domain of large-scale distributed client-server systems such as this.

3.2 Static Analysis

Static analysis was largely accomplished through the use of Microsoft Code Contracts, which feature a number of different checks:

- *Precondition checks:* Ensures the enclosing method satisfies the conditions provided before being run
- *Postcondition checks:* Ensures that certain conditions are met once the enclosing method completes
- *Invariant checks:* Checks that throughout the code a variable always satisfies the specified conditions
- *Assertions:* Checks that, at the moment of the check, the specified condition is met
- *Assumptions:* Checks during runtime that a condition holds- useful if the condition cannot be checked statically (e.g. due to external library calls)

The static analysis tools can be configured to run at compile time, which ensures that the code is regularly analysed. The benefit of using static analysis is twofold: first of all, because the code does not need to be executed to find flaws, static analysis techniques can unearth potential errors that may not appear through conventional dynamic testing (where the conclusion of the tests can only be "The code has failed to break thus far" rather than "The code is unbreakable"); secondly, because the checks can be configured to only run during compilation they do not impact execution time.

The static analysis checks have largely been used in this project as a means of detecting potential null-reference exceptions in the communication protocol and in the key translation of incoming messages to in-game actions. This is due to the large number of conversions from object IDs to in-game objects that take place, as the client will typically send the ID of the object they wish to affect as a string as part of their message. This raises a lot of potential for clients to either include null strings, or to request objects which do not exist.

In terms of determining where best to include static checks, ReSharper's code analysis tools were extremely useful. The depth of analysis can be configured to show warnings wherever

possible null-reference exceptions (or other basic code violations) exist, making it clear where additional checks may be needed. This reduced development time while also catching any areas that were under-restricted.

While static checking is a useful tool in ensuring code correctness, it is best used before writing the code itself. This allows developers to see how code should perform while writing it, and write code that meets the specification. By contrast, writing the static checks after writing the code encourages developers to write them to fit the code. These tests do not provide any useful feedback during analysis, as they are designed to pass. As this project worked with pre-existing code, it was necessary to add the static checks after writing; while it was attempted to make certain that the tests reflected the requirements of the project rather than the code, a preferable approach would have been to build the project with static checks from the start.

3.3 Dynamic Analysis

Dynamic analysis was conducted by creating a series of tests using the Microsoft Unit Test framework which would involve a dummy client attempting to connect and send requests to the server. The unit tests were written in this way as this closely resembles how the end-users will be interacting with the server. The method of testing used was white-box testing, where the internal structure of the code is known and influences tests, as this enabled verification of the state of the server as well as verification of the responses from the server.

The key difference between static and dynamic analysis is that while static analysers attempt to make assertions about the range of possible values variables can be within a method, and make recommendations based on these, dynamic analysis requires the code to be run. In dynamic analysis through unit testing, each test is composed of a mixture of program code and assertions, which determine whether some condition has been met. Tests that do not meet these assertions, or do not complete, are failed tests and indicate a potential problem within the code (or within the test itself).

In this project, a range of actions that could be taken by the client were selected as a basis for the tests. These actions were chosen for being more complicated or (in the case of logging in)

to be integral to security, and covered the main categories of actions: household management, army management, subterfuge, and fief management. The actions under test were logging in, attacking, spying, recruiting troops, maintaining an army, adjusting expenditure, and travelling.

As chapter 4 will cover, this method of testing was work-intensive and time consuming; however, a wide variety of bugs were detected using this technique.

3.4 Security

3.4.1 General Security Principles

In the context of online multiplayer games, it is important to realise that the game client is effectively out of the control of developers as soon as it is released. This means the game client is not part of the trusted code base, and the server cannot rely on the client being correct. Players can and will attempt to decompile and modify it, either out of curiosity or for some personal gain. For this reason, it is imperative that players **cannot** make any assertions about their own state. An example of this is the "Bad Sport" points system in Grand Theft Auto Online, where malicious actions (for example, destroying the vehicles of other players) accrue points that result in being forced to play with players of a similar disposition. Because the points were stored on the client side, players soon discovered a means of resetting the points to zero and as a result could commit malicious actions with impunity.

On a similar note, input from the client should not be trusted. Assume that any and all input is possible and validate it accordingly. In the case of the JominiEngine this includes checking incoming object IDs are non-null, lack special characters, are of the correct format and exist in the master lists of object IDs.

With any service where sensitive information may be involved, it is important that the information is neither sent nor stored as plain text. The reason for this is simple: information can be intercepted during transit or obtained if a server is compromised, and so must be protected. While the JominiEngine currently does not send or store any sensitive information, it is

worth noting that other players may attempt to gain an advantage by intercepting their rival's messages.

3.4.2 Authentication

This subsection describes password security techniques. Hashing refers to using an algorithm to map input data to output of a fixed size (where one can reliably retrieve the output from the input, but not the other way around), while salting refers to prepending a random string of bytes to the beginning of data in order to obfuscate it, and is often combined with hashing.

There are two authentication phases in the JominiEngine: server authentication and client authentication.

During server authentication the server sends its own X509 certificate to the client (along with a session and password salt, to be used in client authentication as described in the paragraph below). The client then attempts to validate this certificate and either continues the log in process by sending its own credentials, or terminates the connection. This method ensures the client can correctly identify the server, reducing the chance of a malicious attacker being able to masquerade as the server.

When a user is registered (which is not covered in this project), their password is hashed with a randomly generated salt (referred to here as the password salt) and stored in the database along with the salt used to create it. When logging in the server sends the client their password salt and a second randomly generated salt (referred to here as the session salt). The client hashes their password with the password salt using the SHA256 hashing algorithm, then hashes the result with the session salt using the same algorithm. The server receives the client's hash and validates it by comparing it to the hash of the session salt and the password hash from the database; if they match, due to the low collision rate of hashing algorithms the server can be reasonably sure the password entered was correct.

The above can be better summarised using the following equations:

$$phash = h(pass, ps)$$

$$hash = h(phash, ss)$$

Where $pass$ is the password, ps is the password salt, ss is the session salt, and $h(a, b)$ is the hashing function applied to b prepended to a.

3.4.3 Secure Connection

While sensitive information will not be transmitted as part of regular gameplay, messages should be protected from being read or altered by anyone other than the intended recipient. To accomplish this the messages are encrypted using the AES symmetric encryption algorithm, which is fast and widely used. This ensures messages are kept secure without incurring a severe performance penalty.

3.4.4 Key Exchange

In order for the messages to be encrypted symmetrically, it is necessary for the client and server to share a key. While there exist several key exchange algorithms, this project uses public key encryption to safely transmit a key from the client to the server.

After the client receives and verifies the server's certificate, it generates a key for use in symmetric encryption. It then uses the public key from the certificate to encrypt the symmetric key and sends it to the server, along with its own login credentials. Upon receiving, the server can decrypt the symmetric key using its private key. This enables the client and server to make use of faster symmetric algorithms without the need to send additional messages for exchanging keys.

3.4.5 Potential Security Flaws

The majority of security concerns stem from potential flaws in the X509 certificate protocol. As pointed out by J. Kim et al (Kim, Choi, and Ryou, 2010), many of these stem from either user carelessness in protecting the certificate's private key, or poorly implemented checks that do not fully verify the authenticity of the certificate. As the certificate is used both for authenticating the server and for securely transmitting the symmetric encryption key, a compromised

certificate is disastrous for the security of the JominiEngine. If the private key of the certificate becomes known, or the certificate is able to be successfully forged, anyone could masquerade as the server and potentially intercept messages from the client.

Kim et al propose some ways to mitigate these security risks. Disconnecting on certificate verification failure, rather than alerting the user and prompting them to disconnect, can reduce the effect user carelessness has on security. Additionally, they recommend that the client must be able to check the certificate's revocation status in order to determine whether the certificate is valid. That way, in the event the server's private key is leaked the signing authority can revoke the certificate and prevent clients from validating it. Both of these checks should be implemented on the game client.

The certificate should also avoid a SHA-1-based signature, as there is a known vulnerability (Stevens, Karpman, and Peyrin, 2015) in SHA-1 which significantly increases the chance of an attacker being able to generate a certificate with the same signature. The certificate signing in the JominiEngine has been implemented using the SHA-256 hashing algorithm, which does not have the same vulnerability.

3.5 Consistency

While it was not originally intended to include any consistency checks or controls in the project, during testing it became apparent that implementing consistency controls on the queue of messages for each client would be necessary.

During testing in certain circumstances a null message would be obtained from the queue of messages. As the method for obtaining messages should either return the next message in the queue or wait for a new message to be received, this ought to have been impossible. However, if two methods called the GetReply() method to obtain the next message from the user, both would attempt to claim the next incoming message and only one would succeed, with the other returning null.

To prevent this, a subclass of the ConcurrentQueue class in .Net was created which included an EventWaitHandle. The ConcurrentQueue class is designed for concurrent operations on a queue and handles all consistency issues, while the EventWaitHandle can be used to signal to threads when an event has occurred. Combined, this presented a thread-safe solution where it was possible to add and remove items from the queue without worrying about consistency issues while also being able to wake up threads when a new message was available, preventing busy waiting. Using this in conjunction with locking ensured no two threads could retrieve the next reply at the same time, and that enqueues and dequeues would not interfere with each other, which solved the concurrency issue in the message queues. This illustrates how classical techniques from operating systems can be used to improve the security of multi-threaded code on the server side.

The focus for this project was on securing the communication protocol, rather than ensuring the consistency of the server side code. However, as a result of switching to an multi-threaded, session-based implementation it is highly likely that there remain many race conditions and other concurrency issues in the code. This will require rigorous testing and repairing in the future, but is out-with the scope of the project.

3.6 Refactoring the Game Engine

Stateless to Session-Based

The previous implementation of the JominiEngine was entirely stateless in terms of the communication protocol: each action relied only on the message that was received and not any before or after. This was largely due to the difficulty in creating session-based communication protocols, as it requires the use of asynchronous tasks. While this was sufficient at the time, it restricted future development in that it is impossible to keep track of the state of the client. In this project a session-based communication protocol has been implemented, where the server can wait asynchronously for replies from the client. This allows much more complicated sequences of messages to be built up, as messages received from the client can be considered part of a longer conversation between client and server.

It is important to note that this is related to session types, but is not the same: a session-based communication protocol infers that the state of each client is kept track of in a session, whereas session typing refers to using types to ensure that the order and type of all messages sent and received is consistent with the communication protocol.

Busy Waiting

The server originally polled for messages non-stop using a while-loop, wasting time by polling even when there were no messages- a process referred to as "busy waiting". To improve performance, and therefore scalability, the server now uses callbacks (more specifically, Lidgren's MessageReceivedEvent) to wait before reading messages.

Modularity

Client messages were originally evaluated entirely within a single case-switch, which grew to thousands of lines of code as the number of actions available to clients increased. To improve readability and maintainability, the logic for each individual action was extracted and placed into separate methods. The case-switch is now used only to determine which action to carry out, making it much easier to manage and change.

3.7 Administrative Components

LogInManager

While previously the LogInManager handled salt generation and hash verification, the user names and hashes were stored in the Server class instead. This has been moved to the LogInManager in order to ensure the LogInManager is a cohesive unit. In the future it should be possible to have a dedicated log-in server in order to reduce strain on the game server, and keeping all log-in related code within LogInManager should make this easier.

ID Validation

Going through the JominiEngine code, it became clear that there was a great deal of repetition when it came to validating an object ID and retrieving the relevant object from the relevant game object list. By creating methods that would take an ID as a string and return either the relevant object or null and an error code, it was possible to reduce the lines of code and improve the code readability as well as maintainability (as shown in Chapter 4.5).

```
case NetIncomingMessageType.ConnectionApproval:
    {
        string senderID = im.ReadString();
        string text = im.ReadString();
        Client client;
        Globals_Server.Clients.TryGetValue(senderID, out client);
        if (client != null)
        {
            ProtoLogIn logIn;
            if (!LogInManager.AcceptConnection(client,text, out
                logIn))
            {
                im.SenderConnection.Deny();
            }
            else
            {
                NetOutgoingMessage msg = server.CreateMessage();
                MemoryStream ms = new MemoryStream();
                // logIn contains certificate, signature and salts
                Serializer.SerializeWithLengthPrefix<ProtoLogIn>(ms,
                    logIn, PrefixStyle.Fixed32);
                msg.Write(ms.GetBuffer());
                lock (ConnectionLock)
                {
                        clientConnections.Add(im.SenderConnection,
                            client);
                        client.connection = im.SenderConnection;
                        client.ProcessConnect();
                }
                im.SenderConnection.Approve(msg);
                server.FlushSendQueue();
                Thread clientThread = new Thread(new ThreadStart(
                    client.ActionControllerAsync));
                clientThread.Start();
                Globals_Server.logEvent("Client␣"+client.username+"␣
                    logs␣in␣from␣"+im.SenderEndPoint);
            }
        }
        else
        {
            im.SenderConnection.Deny("unrecognised");
        }
    }
```

FIGURE 3.1: This code shows the connection approval process for clients wishing to connect to the server

```
Task<ProtoMessage> GetMessageTask = GetMessage();
if (!GetMessageTask.Wait(3000, _linkedTokenSource.Token))
{
    // Taken too long after accepting connection request to receive
       login
    Server.Disconnect(connection, "Failed␣to␣login␣due␣to␣timeout");
    return;
}

ProtoLogIn LogIn = GetMessageTask.Result as ProtoLogIn;
if (LogIn == null || LogIn.ActionType != Actions.LogIn)
{
    // Error- expecting LogIn. Disconnect and send message to client
    ctSource.Cancel();
    Server.Disconnect(connection, "Invalid␣message␣sequence-
       expecting␣login");
    return;
}
else
{
    // Process LogIn
    if (!LogInManager.ProcessLogIn(LogIn, this, _linkedTokenSource.
       Token))
    {
        // Error
        Server.Disconnect(connection, "Log␣in␣failed");
        return;
    }
}
```

FIGURE 3.2: This code shows process of authenticating clients, asserting that a log-in message has been received and verifying the credentials

Chapter 4

Results

This chapter will cover the results of testing and evaluating the various implementations of the JominiEngine.

4.1 Evaluation Strategy

The test run for measuring time and memory was run ten times per implementation, and the results of the tests were averaged excluding the maximum and minimum values.

The time taken was measured from the test client sending a request to the test client receiving a response for that request, and was measured by recording the start and end times in milliseconds using 'DateTime.Now.TimeOfDay.TotalMilliseconds', and then subtracting the end from the start. While this would usually be done with the 'System.Diagnostics.Stopwatch' class, the stopwatch is not threadsafe and produced inaccurate results during testing.

The maximum amount of memory was measured using the 'GC.GetTotalMemory' method. While there are many different ways in C# of measuring the different types of allocated memory, such as paged memory and working set, the 'GC.GetTotalMemory' estimates the total allocated memory and so was deemed the most appropriate. To get the maximum memory, the memory was recorded at the start and end of the test run, as well as after each test. While this is not as accurate as using a profiling tool, as both the Visual Studio performance tools and ReSharper performance tools failed this was selected as the next viable option.

Code maintainability was measured using Visual Studio's code metrics tools (see Chapter 1.3.3).

The number and nature of bugs found during unit testing and static analysis were recorded, and have been summarised and categorised below.

4.2 Error Detection and Prevention with Dynamically-Checked Session Types, Static Analysis and Dynamic Analysis

This section discusses the various software issues encountered during static and dynamic analysis, as well as the issues that could be prevented with session types. Categorisation of issues is based on the IEEE standard definitions for anomalies ("IEEE Standard Classification for Software Anomalies" 2010). Specifically, behaviours that are inconsistent with the requirements (but do not cause the system to crash) are categorised as *defects*, whereas issues that caused the JominiEngine to cease functioning are categorised as *failures*. Additionally, issues that occurred due to concurrency issues and which may only appear in certain circumstances are categorised as *Heisenbugs*, a jargon term and play on words which refers to issues that seemingly disappear when attempts are made to observe them. Revell's article (Revell, 2015) describes Heisenbugs in more detail.

Dynamically-Checked Session Types

The goal of session types is to ensure the correctness of the communication protocol, rather than the correctness of the code itself. As such, this section will focus on how effective dynamically-checked session types were at ensuring the correct order and type of messages.

In the stateless implementation, a mistake in the way clients were added to the list of logged in clients resulted in a defect where clients were able to perform actions before logging in. While the same mistake was present in the session-based implementation, because the order of messages was enforced the client was still incapable of performing actions before logging in. While this defect was prevented, the non-logged in client would still have been able to receive updates from the server. In this case, while one defect was prevented, the underlying issue was not. It could be argued that preventing this defect actually made the issue worse, as it obscured

the underlying problem; however, as mentioned session types are being evaluated here for there effectiveness in ensuring the correct communication protocol, and they did succeed in this.

A client logging in with different credentials on the same connection produces another defect which is prevented with dynamically-checked session types. In the stateless implementation, logging in a second time with different credentials succeeds, resulting in both clients technically being logged in (although only one will be able to send messages at a time). This is undesirable behaviour and could lead to other issues when attempting to send messages to the clients. In the session-based implementation, the second log in request is handled as an invalid request and an error is sent to the client, preventing the second log in. This is due to this implementation being 'aware' that a log in has already taken place, and attempts to process the message as a standard in-game action. Because the action type "LogIn" is not a recognisable action in the game's ActionController, the server sends an error to the client informing them that this action is invalid.

Another defect present in the stateless implementation (though not inherently part of the communication protocol) is the absence of time-outs. While a connection can still time-out if the server does not send or receive from the client for an extended period of time (e.g. due to unexpected network issues), the client can still send 'heartbeat' messages (messages whose sole purpose is to let connected peers know the sender is still connected) to keep the connection alive. By contrast, in the session-based implementation it is possible to specify the time-frame within which the next message from the client must be received, allowing clients who have remained idle for a long time to time-out due to inactivity and freeing-up the connection pool.

While dynamically-checked session types were effective in preventing several issues with the ordering of messages, due to the apparent lack of issues with the types of messages received on the server there was little way of determining whether they were effective in preventing invalid message types from being received. This is due to the nature of how requests are written in the JominiEngine: each request takes the form of a serialized ProtoMessage object, which has an action type, a response type, a string (for an additional message) and an array of strings (for additional information). Requests and responses that require more information

extend this class, and all requests are serialized and de-serialized as ProtoMessage. This is later cast to the appropriate type, and the code for doing so is dependent on the type of action specified in the message. This code has not been modified between the implementations, and has not caused any issues.

Static Analysis

As figure 4.1 shows, the entirety of problems detected were failures, rather than defects. This may be partly due to weak preconditions, but is also in part a weakness of the static analyser.

While many of the bugs detected during static testing were also detected by the dynamic analysis, two were not. One of these was for an action which was not tested dynamically (unbarring characters), while the other was something that was not considered during testing. The latter is a potential failure that can occur if the client somehow overwrites the first byte of the incoming message on the server, which the networking library (Lidgren) uses to store the connection status. If the user was successful in setting this to a value outside the range of the NetConnectionStatus enum (and given the nature and skill of the potential users, this is a possibility), the server would crash. Preventing this is fairly simple matter of checking that the value is in range, but it is not something most testers would even think to test for.

The static analysis tools used were incapable of detecting the presence of Heisenbugs. Given the nature of these bugs, this is unsurprising: the static analyser used here does not check thread scheduling. The analysis tools were however very useful in indicating potential sources of error that would be often overlooked in conventional testing. Unfortunately, analysis also brought up many false-positives, which had to be studied in detail to determine whether or not they could be potential sources of error. In spite of this extra effort, static analysis is recommended as a complimentary method of testing to conventional unit testing, as the some of the potential failures detected were outside what would be covered in a typical unit test.

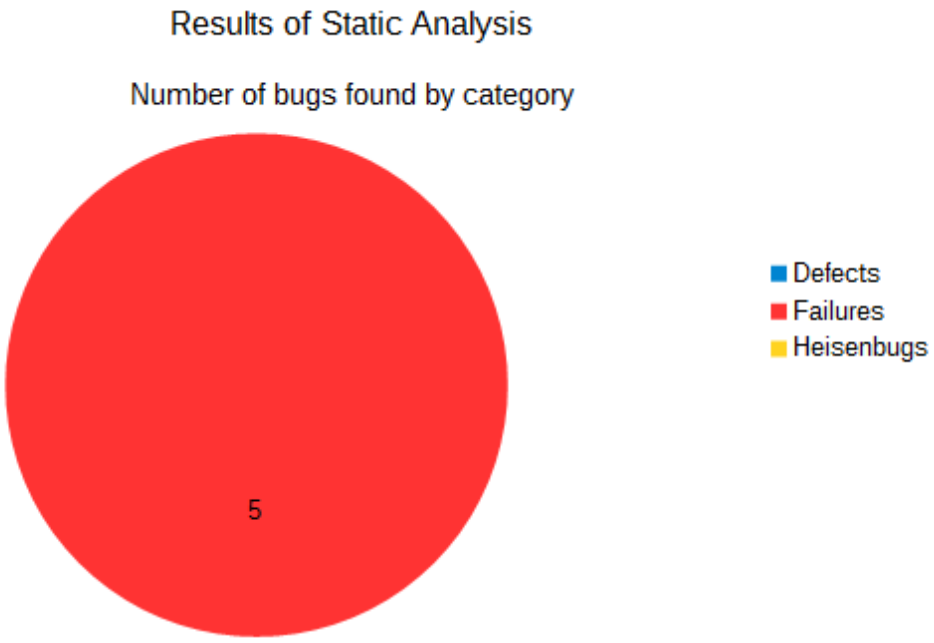


FIGURE 4.1: Chart of the number of bugs found by category during static testing

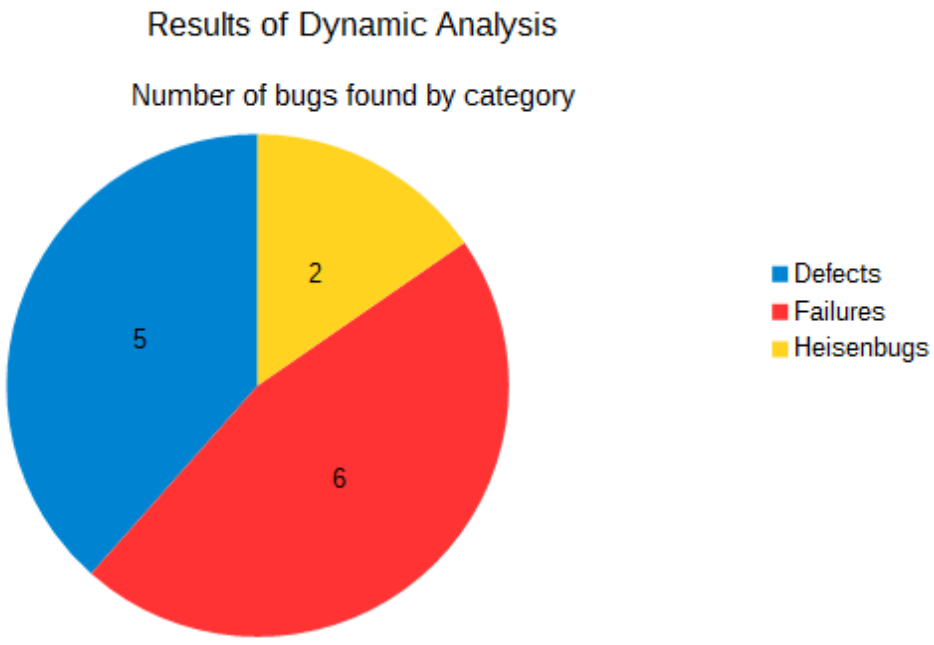


FIGURE 4.2: Chart of the number of bugs found by category during dynamic testing

Dynamic Analysis

As illustrated in figure 4.2, dynamic analysis found a much broader range of bugs compared to static analysis (though the majority of these were still failures).

Dynamic analysis was the only method of analysis that detected Heisenbugs, though this was by chance rather than by design. The first Heisenbug was found when once call to method for retrieving the next message from the client returned null. This was previously thought to be an impossibility; the method was intended to wait for the next message, and would be cancelled if the client disconnected or took too long to reply. After adding some debugging statements to the code, the bug disappeared- re-appearing when the debugging statements were removed. It was eventually determined that if two tasks were waiting for the next message, a single message was received, and both attempted to claim the message at the same time, one would succeed and the other would return null. This was later fixed by using ConcurrentQueues, which are thread safe, and locks.

The second Heisenbug was detected not during a unit test, but during the test run designed to measure time and memory performance. The bug essentially involves race conditions on the dictionary mapping connections to clients: if a disconnect is interleaved with a connect, the server will attempt to search the list of connected clients with an empty connection. This results in a failure, and was solved without much difficulty using locks. While other tests had involved clients logging out and then logging in again, the bug did not occur in any of those tests, surfacing only when it was least expected.

The majority of defects detected were due to logging in and logging out not being handled correctly in the stateless implementation. These included the server still continuing to process a client's message after the client logs out, no time outs, and the client being able to log in twice with different credentials on the same connection. These were detected through failed unit tests.

Dynamic analysis was capable of finding a wide range of bugs - some of which were not being tested for - and was an effective means of determining under which conditions a method would fail. However, as mentioned in the static analysis section above, there were failures that

dynamic testing did not detect simply because it was not obvious that they could occur. While dynamic analysis and unit testing should remain an integral part of software testing, it is best complimented with static analysis to get more in-depth coverage.

4.3 Impact of Session-based Implementation on Performance

This section compares the performance of implementing a session-based communication protocol to the performance of the stateless communication protocol, in terms of both time and maximum memory consumption.

Time

Figure 4.3 shows the time performance of the various implementations. Comparing the time taken for the encrypted session-based and stateless implementations (blue and orange), as well as the time taken for the unencrypted session-based and stateless implementations (yellow and green), it is apparent that the session-based implementation takes longer in every action. This is most noticeable in LogIn, where the session-based implementation takes around two-hundred milliseconds longer. In total, there was a 36% increase from the encrypted stateless to session-based implementation, and a 32% increase from the unencrypted stateless to session based implementation. While this is a fairly large increase, it is worth noting that the time for Recruit and Spy includes the time taken to receive the initial message stating the cost of recruiting and success chance for spying, and the time taken to receive the result after sending the confirmation. Additionally, as logging in is a once-per-session action, an increase in this time will have only a small impact on the performance of the server. The increase in time in the Recruit, Travel and Spy actions is small - less than twenty milliseconds on average in all cases - and so should not have a noticeable negative affect on the performance of the server.

Memory

Figure 4.4 shows the impact various implementations had on the maximum total allocated memory. Due to aforementioned issues with memory profilers, the memory measured is not

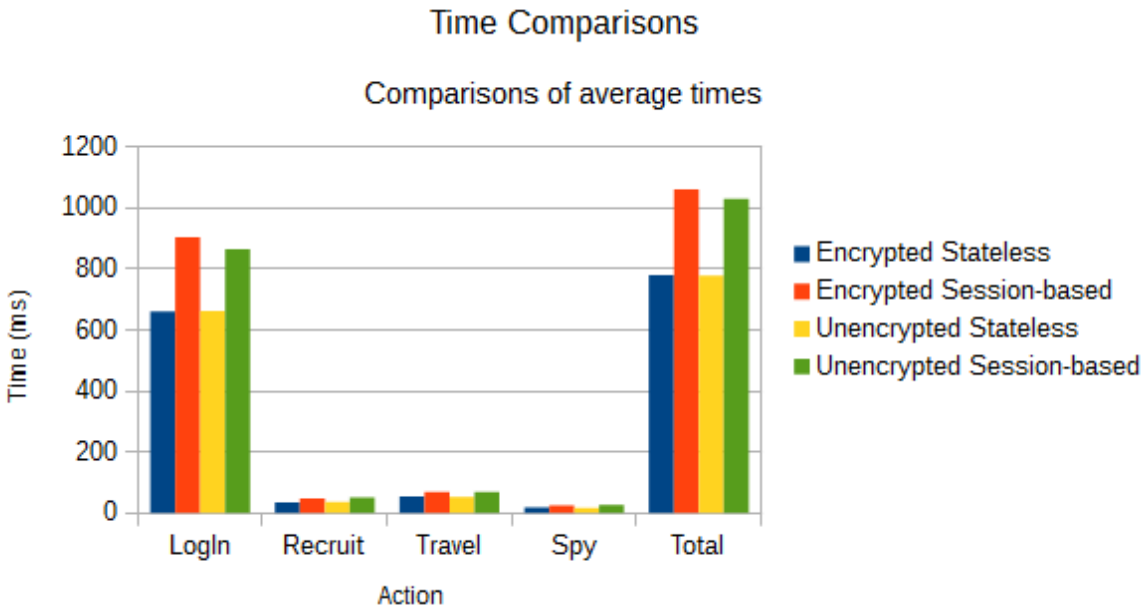


FIGURE 4.3: Time comparisons of stateless and session-based implementations, both encrypted and unencrypted

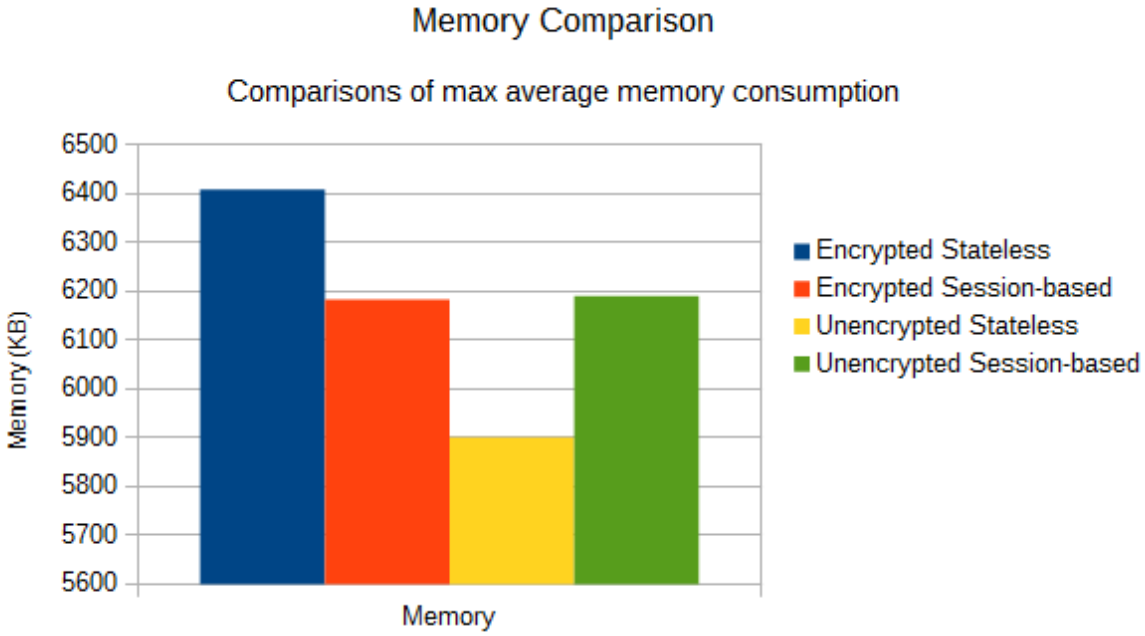


FIGURE 4.4: Memory comparisons of stateless and session-based implementations, both encrypted and unencrypted

exact and fluctuated between tests.

Comparing the encrypted stateless implementation to the session-based implementation, the session-based implementation appears to use less memory; however, comparing the two unencrypted implementations shows the session-based implementation using more memory. Based on this, it can only be assumed that the session-based implementation has little impact on memory consumption.

4.4 Impact of Encrypting Messages on Performance

This section compares the impact encrypting messages has on the time and memory performance of the various implementations.

Time

Referring back to the recorded times in Figure 4.3, the times of the encrypted and unencrypted stateless implementations (blue and yellow) and the encrypted and unencrypted session-based implementations (orange and green) are now compared.

In the stateless implementation, encryption has no noticeable affect on time throughout all actions. As appendix A.1 shows, the average total completion time is approximately eight milliseconds more for the encrypted version- small enough to be negligible.

In the session based implementation, there is a noticeable difference between the encrypted and unencrypted implementations- particularly while logging in. In total, the encrypted implementation took 3% longer to complete- however, this masks the underlying results. Referring to Appendix A.1, the time taken to log in is 4% longer in the encrypted implementation versus the unencrypted implementation. The time taken to perform recruitment, travelling or spying on the other hand, is actually *less* with encryption than without encryption. The increase in log in time can be explained by the required set-up of the shared encryption key. The source of the decrease in the recruit, travel and spy times is less obvious- as the decrease is present across all actions, it is unlikely to be accidental. It might be encrypted messages

are more efficient to send and receive, though without performing more in-depth tests this remains unclear.

Memory

Referring back to figure 4.4, this section compares the encrypted and unencrypted stateless implementations (blue and yellow) and the encrypted and unencrypted session-based implementations (orange and green).

While there is little change in memory consumption for the encrypted and unencrypted session-based implementation, the difference between the encrypted and unencrypted stateless implementation is much greater. This is likely due to aforementioned issues with measuring the memory consumption, as the difference in memory is too large to be entirely due to encryption. Furthermore, the session-based implementation shows no such leap in memory, reinforcing the theory that this is accidental. From these results, it can only be determined that encryption has little or no impact on memory consumption.

4.5 Changes in Code Complexity and Maintainability

This section examines the changes reported by Visual Studio's code metrics (see Chapter 1.3.3) across the different versions of the project. Here the "Original" implementation refers to the project before refactoring took place, and is mostly used to show how refactoring improved the maintainability of the code. "Stateless" refers to the implementation that does not use sessions and dynamically-checked session types in the communication protocol, whereas "Session-based" refers to the implementation with them.

Changes to the Server Class

As shown in figure 4.5, the main changes between the original and stateless implementations were a slight increase in complexity and coupling, and a slight decrease in lines of code. The

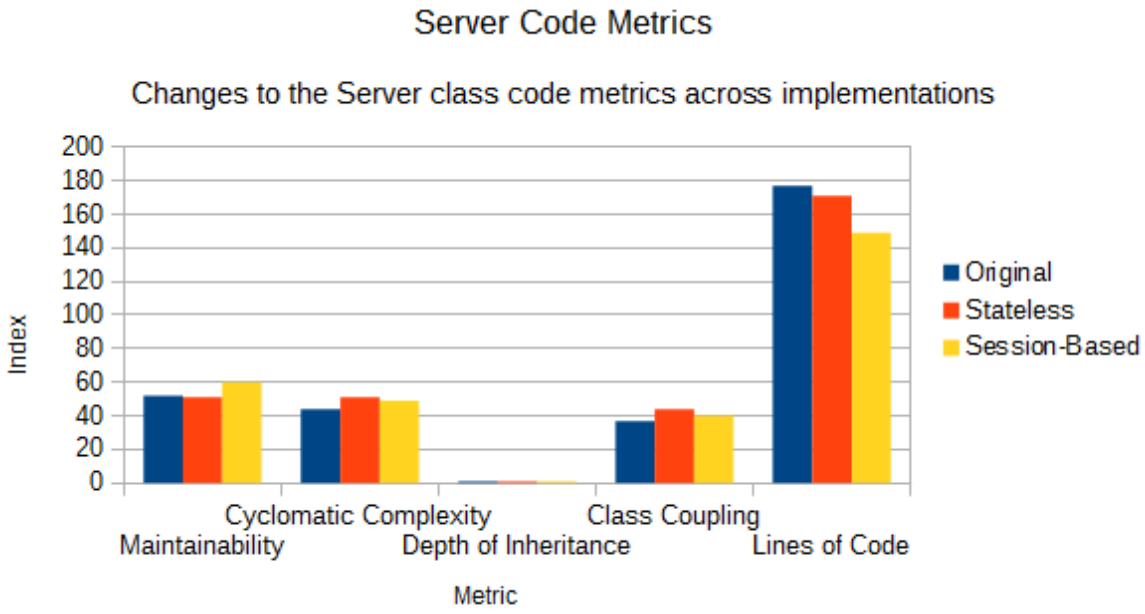


FIGURE 4.5: Changes to the code metrics of the Server class across implementations

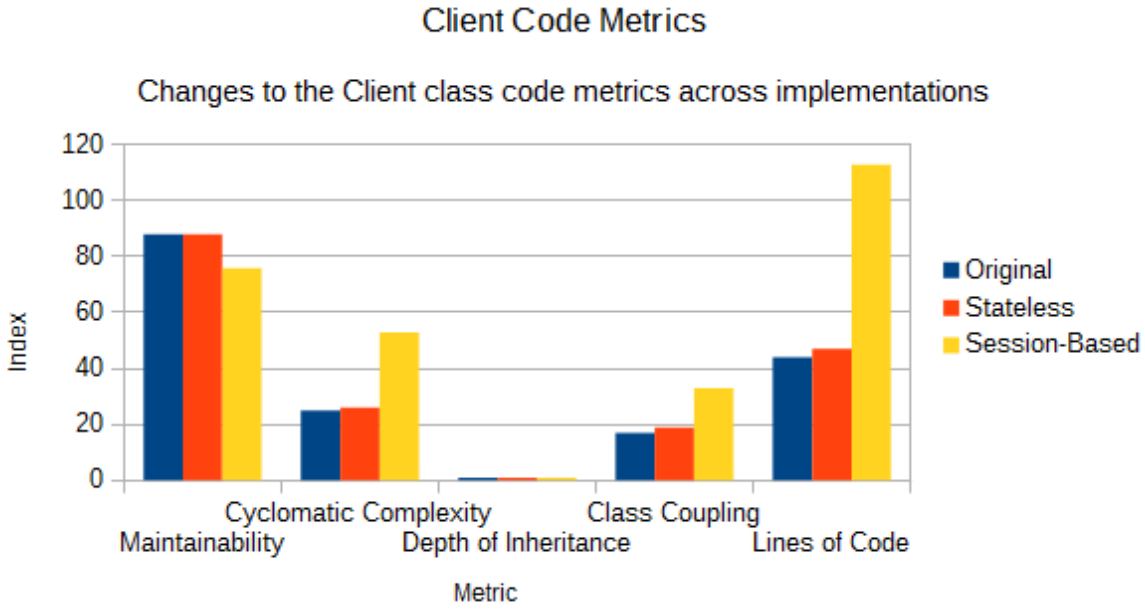


FIGURE 4.6: Changes to the code metrics of the Client class across implementations

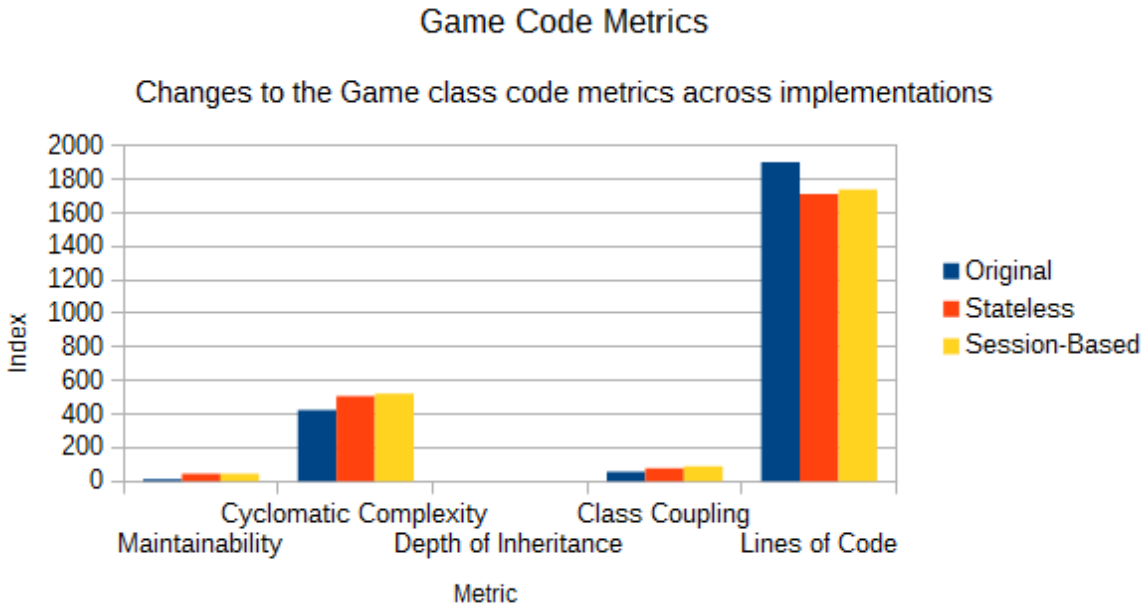


FIGURE 4.7: Changes to the code metrics of the Game class across implementations

changes in complexity and coupling were a result of moving the verification of client credentials and the log in process from the Server class to the LogInManager class, decreasing the amount of code but increasing coupling. Additional parameter checks were behind the increased the cyclomatic complexity. While the code maintainability stayed roughly the same, moving the log in code to the LogInManager class was an improvement in terms of class design as the LogInManager should be the sole class responsible for all authentication and log in functionality.

The changes from the stateless to session-based implementations are largely due to the code for message processing being moved from the Server class to the Client class (which is also reflected in the increase in lines of code in figure 4.6). This resulted in a small increase in maintainability, as well as a decrease in cyclomatic complexity, coupling and lines of code.

Changes to the Client Class

Figure 4.6 shows the different code metrics for the various implementations of the Client class. While there were very few changes between the original and stateless implementations, the session-based implementation changed dramatically in terms of maintainability, complexity, coupling and lines of code.

In order to obtain the next messages from the client, several additional methods had to be added to asynchronously retrieve message from a queue (refer to section 3.1 for more details). Moving the message processing logic from the Server class to the Client class further contributed to the increased coupling, lines of code and complexity, while the use of dynamically-checked session types required additional code to ensure the correct order of messages, further increasing complexity and lines of code.

In spite of this, the maintainability index only dropped 8 points from 88 in the stateless implementation to 76. Considering that this is a fairly drastic change in the communication protocol, the slight reduction in maintainability can be taken as a sign that this change has been implemented in a clear, maintainable way.

Changes to the Game Class

This section examines the changes in code metrics in the Game class across the various implementations.

Looking at figure 4.7, it is clear that the maintainability index is quite low. The class is, by nature, hard to maintain, as it handles the set-up for the game as well as validating and processing client's requests to carry out actions. This depends on the current structure and contents of each type of request, the rules associated with the request, and the methods that will be carried out if the request is successful.

While the maintainability index is still low in the session-based and stateless implementations, what the chart does not show is the impressive increase in maintainability from the original implementation. As appendix B.3 shows, the maintainability index increased from 13 in the original implementation to 46 in the stateless and session-based implementations-

an increase of almost 254%. This was largely due to refactoring Game's ActionController as described in Chapter 1.6.

Visual Studio's code metrics for the ActionController method can be seen in Appendix B.4, which assigned the original implementation's ActionController a maintainability index of 0 and a cyclomatic complexity of 356, indicating that it was essentially unmaintainable. After refactoring, the new index increased to 11 and the cyclomatic complexity had been reduced by over 68% to 112. Refactoring also reduced the case-switch to 153 lines of IL code (approximately 470 lines of C# code), which (while still difficult to maintain) is an improvement over the original.

Appendix B.3 also shows no decrease in maintainability index between the stateless and session-based implementations. This can be taken as an indication that the session-based version of the Game class is not any more difficult to maintain.

4.6 A Subjective View on the Relative Difficulty of Implementation

Sessions and Dynamically Checked Session Types

Transitioning from a sequential, stateless communication protocol to a concurrent, session-based one was a huge jump in complexity, though not due to dynamically checked session types. Dynamically checking the order and type of messages was comparatively easy, as it could be done by waiting for the next message, checking the type of client request and attempting to cast to the expected type. The main difficulties lay in consistency controls, event handling and task cancellation.

To give a clearer picture of how messages are processed in the stateless implementation, here is a brief summary: When the server is started a long-running thread is created, which has the sole purpose of receiving incoming messages and deciding what to do with them. When a message is received the thread processes it synchronously, then retrieves the next one. When there are no more messages, the thread waits until it receives a MessageReceivedEvent, and retrieves the new message. This thread will continue until the server is shut down.

In the session-based implementation, there is still the long-running server thread as described above. However, when a client connects a new thread is spawned to handle this client's requests (referred to here as a client thread). Much like the server thread, each client thread runs until the client disconnects. The client thread waits for a LogIn message, processes it and, if successful, repeatedly waits for and processes the next request from the client. Each Client object in the server has their own queue for incoming messages. When the server receives an incoming message from a client, it is added to this queue. This fires an event, waking up the client thread and causing the message to be retrieved. In brief, we have a long-running thread that spawns other long-running threads that create asynchronous tasks.

Here lies the problem: When the client disconnects, or the server shuts down, all tasks for the affected clients must cancel and all threads must end. Additionally, if there are multiple tasks waiting for the next message from the client, only one should be able to get the next message while the others continue to wait- although when a message is received all of the waiting tasks will wake up and attempt to claim it. To handle all this correctly required the use of event handlers, locking, and linked cancellation tokens. This is rather complicated, particularly for a solo project, and especially for those inexperienced in asynchronous and concurrent programming. My opinion is that dynamically checking session types is feasible, but transitioning to a session-based implementation is not recommended for the inexperienced or those working alone unless using a library that provides the above functionality.

Dynamic Analysis with Microsoft Unit Test Framework

Unit testing will be familiar to the majority of developers, as it is an established method of testing and debugging functions. In this project Microsoft Unit Test Framework was used not for testing units of code, but client actions: Each test had the test client attempt to send a request with various parameters, and validated both the reply received from the server and, in some cases, the state of the game after processing the action. This method of testing, while not as thorough as unit testing, was much closer to real exchanges between the server and clients.

The Microsoft Unit Test Framework was not especially difficult to use; being well-documented and having a few key functions and attributes (such as Assert.IsTrue(....)), it took very little

time to learn and was expressive enough to provide the necessary control over tests required (such as having time-outs on test methods, and being able to set-up the test environment using ClassInitializeAttribute).

The one difficulty in this method of dynamic analysis is the amount of work involved: as the majority of parameters could not be automatically generated (as much of the tests were to do with permissions and object ownership), the tests were written by hand. With over fifty tests and five-hundred lines of code (for each implementation), this was extremely time consuming. Bear in mind this was only to test seven actions (including logging in) out of over fifty - exhaustively testing every action would take even longer.

In addition, during testing I found the tests were often being adjusted to fit the behaviour of the code rather than the other way around, simple because the "correct" behaviour of the code had been misinterpreted or forgotten. To prevent this, it would be beneficial to use a Test-Driven Development (TDD) (Beck, 2002) approach, which focuses on writing tests first. While this was not possible for this project as it builds on existing code, in TDD the tests act as a form of guidance to the developer, as the tests are based on the specification rather than on the code, and so the correct behaviour of the code can be determined by looking at the tests. This also spreads out the test writing, and ensures that testing is not a last minute, often-neglected task.

Static Analysis with Code Contracts

Despite not being a familiar tool, Code Contracts were comparatively simple to use. The majority of work was performed by the static analyser, which could be configured in terms of which potential issues to report, what recommendations to make, and the warning level (from only the most relevant warnings to all warnings). The only effort that had to be expended was in configuring the analyser to focus only on specific classes (which required adding the line 'assembly: ContractVerification(false)' to the AssembleInfo.cs file and then adding the 'ContractVerificaton(true)' attribute to classes to be tested), and adding preconditions, post-conditions and invariants where appropriate. To run static analysis on the main sources of error (the Server, Client, Game, Army and Battle classes) required only adding thirty-one lines to the project.

Perhaps the only difficulties in using Code Contracts for static analysis was in interpreting the suggestions, which were often vague, and the long analysis time. Restricting the analysis to certain classes as above was necessary to ensure the analysis performed in a timely manner. I personally would recommend Code Contracts (or other static analysis tools) to anyone with knowledge of how to use preconditions and postconditions, as they require minimal effort to implement while providing some useful insight on potential sources of error in the code.

Chapter 5

Conclusion

This project has evaluated how effective static analysis, dynamic analysis and dynamically checked session types are in improving the security and correctness of large distributed systems, in addition to how difficult they were to implement, and how the changes to the case study impacted the time and memory performance. This section summarises these results and makes recommendations for the development of distributed systems and, more specifically, MMOs.

Dynamically checked session types were found to be effective in preventing a variety of communication protocol violations, as shown in Chapter 4.2, and improved the security of the case study by ensuring that a client had logged in correctly before the client could perform any authorised actions. Chapter 4.5 shows that the session-based implementation with dynamically checked session types did not reduce the maintainability of the code by any amount that would cause concern; however, as Chapter 4.3 shows, this implementation came with some loss of performance- the test runs took on average over 30% longer to complete. This illustrates the benefits statically-checked session types could have for large scale, distributed systems: with static type checking, there would be much less of an impact on performance while also ensuring the correct implementation of the communication protocol. As dynamically checked session types were effective in improving the security and correctness of the communication protocol, it can be concluded that they are beneficial to large distributed systems- though statically checked session types would be preferable for distributed systems with tight performance constraints.

Static analysis using Code Contracts found a fairly narrow range of bugs, identifying only a

few that would potentially cause failures (as well as several that would never come to fruition, or false positives)- however as Chapter 4.2 notes some of these detected failures were far outside the range of normal testing, and likely would never have been found through unit testing alone. Due to the nature of the case study, the JominiEngine (refer to Chapter 1.4.1 for more on the project history), finding potential failures before deployment is important, as if a flaw is not found during testing, it will be found (and quickly) by the end users after release. In addition, Chapter 4.6 discusses the relative ease of implementing static analysis using Code Contracts, and concludes that Code Contracts are a fast to implement, useful way of testing code. While static analysis has been largely used for the verification of safety-critical systems, drawing from the results of this project static analysis tools are recommended as part of standard project testing- though should not replace dynamic tests, particularly in large, distributed, concurrent systems such as the JominiEngine.

Dynamic analysis using Microsoft Unit Test framework found a wide range of bugs, some of which were not being tested for. The results in Chapter 4.2 show that dynamic analysis located defects, failures and Heisenbugs, and though detection of the latter was largely accidental, it still illustrates the importance of running the code with a wide variety of parameters to locate well-hidden bugs. Dynamic analysis is an established part of the software testing process, and is irreplaceable. However, as described in Chapter 4.6 there is a large amount of work required to set up the dynamic tests. Testing large projects in this way is extremely time consuming and work-intensive, and it may be impossible to exhaustively test every unit and subsystem- particularly in the case of the JominiEngine, where an existing framework of unit tests was unavailable. As such, it is recommended to use a modified approach to Test Driven Development (TDD) that integrates static precondition, postcondition and invariant checking into the tests, as TDD has been shown to improve code quality over other approaches (Bhat and Nagappan, 2006). The static analyser can check for potential exceptions and violations of preconditions and postconditions, which improves the effectiveness of testing when combined with dynamic analysis. Well-defined preconditions and postconditions could also reduce the number of unit tests required to test a method fully, as the static analyser would eliminate invalid parameters.

To improve the integrity and privacy of messages sent between clients and the JominiEngine, messages were encrypted using a symmetric encryption algorithm. While it was previously thought that message encryption would have a noticeable negative impact on performance, Chapter 4.4 shows that this impact was extremely small, having a slight negative impact on time and no discernible impact on memory. As Man in the Middle (MITM) attacks are an established attack that targets the connection between the client and the server, using encryption to prevent messages being intercepted and tampered with is highly recommended for distributed systems, even those with a focus on low response times, as the time taken to encrypt a message is offset by the increased security.

In summary, both static and dynamic analysis are recommended for the testing of distributed systems (including MMO game servers), and a suitable method of employing them would be through TDD. Session types have the potential to improve the security and correctness of distributed systems, but until a mainstream statically checked implementation exists, developers systems that require fast execution may wish to avoid them.

Chapter 6

Future Work

As many of the techniques and technologies used in this project are continually being researched, it is hard to know what new techniques and tools will be available in the near future - particularly when it comes to session types. As mentioned in Chapter 2.1, a hybrid implementation of session types for modern languages has been proposed, and future work could attempt to implement this using the JominiEngine as a case study and evaluate the appropriateness for large distributed systems and MMO game servers. However, this would require a complete re-write of the JominiEngine server, and should not be taken lightly.

As consistency has not been a core focus of the project, only the minimum consistency controls for the correct behaviour of the server (including retrieving client messages and modifying the list of connected clients - see 4.2 for more details) have been implemented. Due to the use of concurrency, the session-based implementation may have introduced further consistency issues throughout the game. Future work could evaluate methods of detecting inconsistencies and race conditions, or preventing them, with a focus on large distributed systems.

While it was initially intended to evaluate how modelling could be used to examine the correctness of the design of large projects, it was later determined that this was largely outside the scope of static and dynamic code verification and so was not directly relevant to the project. Additionally, during attempts to model subsystems of the game it became apparent that the game objects were so closely linked that it was nearly impossible to cover a subsystem in detail (such as the inheritance subsystem) without touching upon several others. In order to model the (fairly complex) architecture of the JominiEngine a huge amount of abstraction was necessary, so much so that the model did not closely resemble the true game objects and

had little benefit to the project. With that being said, model-based verification is an established method of verification, and there is plenty of research on the model-based verification of distributed systems. A future project could examine the effectiveness of these, though it may be better evaluated using a smaller and simpler case study than the JominiEngine.

Appendix A

Runtime Measurements

A.1 Time Measurements

Implementation	Log In Time	Recruit Time	Travel Time	Spy Time	Total Time
Encrypted Stateless	661.253625	34.86125	54.00375	18.12625	780.6485
Encrypted Session-based	905.5175	48.90125	68.53875	25.0125	1062.73
Unencrypted Stateless	663.644	37.505	53.34125	17.5	778.978625
Unencrypted Session-based	865.245	51.655375	68.91375	26.14	1030.9625

FIGURE A.1: The average time (in milliseconds) of ten runs (excluding maximum and minimum values) for each implementation

A.2 Memory Measurements

Implementation	Memory (kilobytes)
Encrypted Stateless	6409.17
Encrypted Session-based	6183.03
Unencrypted Stateless	5900.83
Unencrypted Session-based	6190.69

FIGURE A.2: The average maximum memory consumed (in kilobytse) of ten runs (excluding maximum and minimum values) for each implementation

Appendix B

Code Metrics

Code Metrics: Server

Implementation	Maintainability Index	Cyclomatic Complexity	Inheritance	Class Coupling	Lines of Code
Original	52	44	1	37	117
Stateless	51	60	1	44	171
Session-based	60	49	1	40	149

FIGURE B.1: The code metrics obtained from Visual Studio on the Server class, across three implementations

Code Metrics: Client

Implementation	Maintainability Index	Cyclomatic Complexity	Inheritance	Class Coupling	Lines of Code
Original	88	25	1	17	44
Stateless	88	26	1	19	47
Session-based	76	53	1	33	113

FIGURE B.2: The code metrics obtained from Visual Studio on the Client class, across three implementations

Code Metrics: Game

Implementation	Maintainability Index	Cyclomatic Complexity	Inheritance	Class Coupling	Lines of Code
Original	13	424	1	56	1904
Stateless	46	507	1	78	1715
Session-based	46	523	1	90	1741

FIGURE B.3: The code metrics obtained from Visual Studio on the Game class, across three implementations

Code Metrics: ActionController (Game)

Implementation	Maintainability Index	Cyclomatic Complexity	Class Coupling	Lines of Code
Original	0	356	56	1332
Stateless	11	112	18	153
Session-based	10	113	22	156

FIGURE B.4: The code metrics obtained from Visual Studio on the ActionController method of the Game class, across three implementations

Appendix C

Bug Results

Categorisation of Bugs Found

Detected by	Defects	Failures	Heisenbugs
Static analysis	0	5	0
Dynamic analysis	5	6	0

FIGURE C.1: The number and type of bugs found, as well as which method succeeded in detecting them

References

Abramsky, Samson (1993). "Computational interpretations of linear logic". In: *Theoretical Computer Science* 111.1-2, pp. 3–57. ISSN: 03043975. DOI: 10.1016/0304-3975(93)90181-R. URL: http://www.sciencedirect.com/science/article/pii/030439759390181R.

Anderson, Paul (2008). "The use and limitations of static-analysis tools to improve software quality". In: *CrossTalk: The Journal of Defense Software Engineering*.

Beck (2002). *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0321146530.

Bhat, Thirumalesh and Nachiappan Nagappan (2006). "Evaluating the Efficacy of Test-driven Development: Industrial Case Studies". In: *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*. ISESE '06. New York, NY, USA: ACM, pp. 356–363. ISBN: 1-59593-218-6. DOI: 10.1145/1159733.1159787. URL: http://doi.acm.org/10.1145/1159733.1159787.

Bocchi, Laura, Weizhen Yang, and Nobuko Yoshida (2014). "Timed Multiparty Session Types". In: *In 25th International Conference on Concurrency Theory (CONCUR 2014)*. Springer, pp. 419–434.

Bond, David (2015). "Design and implementation of a massively multi-player online historical role-playing game". MSc Thesis. Heriot-Watt University.

Common Criteria Maintenance Board (2012). *Common Criteria for Information Technology Security Evaluation Part 1 : Introduction and general model*. Tech. rep. CCMB-2012-09-001. Common Criteria.

Deligiannis, Pantazis et al. (2015). "Asynchronous Programming, Analysis and Testing with State Machines". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2015. Portland, OR, USA: ACM, pp. 154–164. ISBN:

978-1-4503-3468-6. DOI: 10.1145/2737924.2737996. URL: http://doi.acm.org/10.1145/2737924.2737996.

Deursen, Arie van (2014). *Think Twice Before Using the "Maintainability Index"*. URL: https://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index/ (visited on 04/18/2016).

Fähndrich, Manuel (2010). "Static verification for code contracts". In: *Proceedings of the 17th international conference on Static analysis*. Springer-Verlag, pp. 2–5.

Gasior, Lukasz (2014). *ReSharper Essentials*. Packt Publishing. URL: https://www.jetbrains.com/resharper/.

Goseva-Popstojanova, Katerina and Andrei Perhinschi (2015). "On the capability of static code analysis to detect security vulnerabilities". In: *Information and Software Technology* 68, pp. 18–33. DOI: 10.1016/j.infsof.2015.08.002.

Honda, Kohei (1993). "Types for Dyadic Interaction". In: *CONCUR'93*. Springer Berlin Heidelberg, pp. 509–523.

Hu, Raymond and Nobuko Yoshida (2016). "Hybrid Session Verification through Endpoint API Generation". In: *FASE 2016*. LNCS. Springer.

Hu, Raymond et al. (2009). *The SJ Framework for Transport-Independent, Type-Safe, Object-Oriented Communications Programming*.

"IEEE Standard Classification for Software Anomalies" (2010). In: *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pp. 1–23. DOI: 10.1109/IEEESTD.2010.5399061.

Johnson, Bruce (2014). *Professional Visual Studio 2013*. 1st Edition. John Wiley & Sons.

Kim, J H, H B Choi, and J C Ryou (2010). "Countermeasures to Vulnerability of Certificate Application in u-City". In: *Ubiquitous Information Technologies and Applications (CUTE), 2010 Proceedings of the 5th International Conference on*, pp. 1–5. DOI: 10.1109/ICUT.2010.5677755.

Klophaus, Rusty (2010). "Riak Core: building distributed applications without shared state". In: *ACM SIGPLAN Commercial Users of Functional Programming*. ACM New York, Article No. 14.

Microsoft Developer Network. *Unit Testing Framework*. URL: https://msdn.microsoft.com/en-us/library/ms243147(v=vs.90).aspx (visited on 04/23/2016).

— (2007). *Maintainability Index Range and Meaning*. URL: https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/ (visited on 04/24/2016).

Musuvathi, Madanlal et al. (2009). "Finding and reproducing Heisenbugs in concurrent programs". In: *In Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pp. 267–280.

Revell, Timothy (2015). "Bugged out". In: *New Scientist* 228.3050, pp. 40–43. ISSN: 0262-4079. DOI: http://dx.doi.org/10.1016/S0262-4079(15)31813-3. URL: http://www.sciencedirect.com/science/article/pii/S0262407915318133.

Sözer, H (2015). "Integrated static code analysis and runtime verification". In: *Software: Practice and Experience* 45.10, pp. 1359–1373. DOI: 10.1002/spe.2287.

Stevens, Marc, Pierre Karpman, and Thomas Peyrin (2015). *Freestart collision for full SHA-1*. Cryptology ePrint Archive, Report 2015/967. http://eprint.iacr.org/.

Wadler, P., N. Yoshida, and S.J. Gay (2013). *Grant: From Data Types to Session Types—A Basis for Concurrency and Distribution*.

Wadler, Philip (2012). "Propositions as Sessions". In: *International Conference on Functional Programming (ICFP)*.

— (2016). *A Basis for Concurrency and Distribution*. URL: http://groups.inf.ed.ac.uk/abcd/ (visited on 04/15/2016).

Yoshida, Nobuko et al. (2013). "The Scribble Protocol Language". In: *8th International Symposium on Trustworthy Global Computing (TGC '13)* October. URL: http://mrg.doc.ic.ac.uk/publications/the-scribble-protocol-language/invited.pdf.