University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Comparing Reliability Mechanisms for Secure Web Servers: Actors, Exceptions and Futures

Danail Penev

March 27, 2019

Abstract

Modern web applications must be secure and use authentication and authorisation for verifying the identity and the permissions of users. Programming language reliability mechanisms are a key technology for handling web app security failures. These mechanisms include actors, exceptions and futures. This dissertation compares the performance and programmability of the three reliability mechanisms for secure web applications. Key performance metrics are throughput and latency for workloads comprising successful, unsuccessful and mixed requests across increasing levels of concurrent connections. Our programmability study focuses on available attack surface.

This dissertations presents a critical survey of reliability mechanisms, the let-it-crash philosophy and cybersecurity. It discusses application-level reliability mechanisms, exceptions and futures. The let-it-crash philosophy is introduced and its relationship to the actor model and its implementations. Finally, cyberscurity in the form of authentication and authorisation is reviewed.

An experiment is designed, focusing on programmability and performance for the three reliability mechanisms in varying scenarios, based on the request types, the number of concurrently open connections and the security action. The core of this experiment is the development of a secure web application benchmark, implementing actors, exceptions and futures.

The results show authentication and authorisation follow similar performance patterns. Exceptions have the best throughput for only successful requests. For only unsuccessful requests, actors have the best throughput at low numbers of concurrent connections, while futures peak at high number of connections. Exceptions have the lowest mean latency for only successful and for only unsuccessful requests. While unsuccessful requests have low mean latency, they dramatically reduce the throughput for all three models, compared to successful requests. In a realistic mixed-request scenario, exceptions have the highest throughput and the lowest latency, followed by futures and actors. In terms of programmability, actors introduce the fewest new states in the program, followed by futures and exceptions. Thus, actors have the least attack surface.

Exceptions have the best performance, but introduce the most new states. In contrast, actors introduce the fewest states but have the worst performance, in terms of throughput and latency. The research shows a clear inverse relationship between performance and programmability in actors, exceptions and futures for authentication and authorisation.

Education Use Consent

Contents

1 Introduction

1.1 Context

Security is a key requirement for modern software applications, as users rely on their devices for flight bookings, applying for documents, etc. Verifying the user's identity through authentication and authorisation is thus done multiple times in a typical interaction. Requiring user login with a password remains a common mechanism, and in web applications like social media user identity and allowed actions can be verified as often as at every request that is being made. Frequent verifications at various points in the interaction increase the potential risk for cyber attacks, as malicious agents have more opportunities for attacks. This raises the need for reliable, robust and secure server applications.

Handling security failures, e.g. when a user fails to successfully authenticate, securely and efficiently is important for software in generals, and web apps specifically. Programming languages offer different reliability mechanisms for recovering from error states. One of the most common methods for dealing with unexpected failures is exceptions. An action that can potentially fail is surrounded by a try-catch block, forcing the developer to take the appropriate precautions. The actor model ensures its reliability through the let-it-crash philosophy. Actors that fail computations are left to die, while an associated supervisor deals with the aftermath. More recently, responsive interfaces and frequent asynchronous calls have seen a rise in the popularity of futures - a paradigm for decoupling a value from its computation. In the future model, both successful and failing outcomes are considered equally likely and follow-up actions are specified for each response. Exceptions, actors and futures offer reliability at different abstraction and programming complexity levels.

The aim of this project is to compare the performance and secure programming implications of three reliability mechanisms for handling security failures in web apps. The three models are compared in terms of their throughput and latency in a web service setting, programming effort required and resulting attack surface available.

1.2 Achievements

This chapter covers the objectives that were achieved in the scope of this project.

Critical Literature Review Chapter 2 presents a critical survey of reliability mechanisms, the let-it-crash philosophy, cybersecurity and their potential interaction. The first part of it is a detailed review of different application-level reliability mechanisms (Section 2.1). It presents a discussion on exceptions and the criticism they have faced over the years. Finally, the future model is introduced. The second part is an introduction of the let-it-crash reliability philosophy (Section 2.2). It then looks in detail at the actor model and its history. Actor implementations in different languages and frameworks are covered. Finally, the literature review looks at cybersecurity in modern applications (Section 2.3). In particular, it focuses on authentication and authorisation. This section discusses the difference between the two security mechanisms and why each of them is necessary.

Experiment Design The experiment is designed to show the performance of different programming language security failure recovery mechanisms, in terms of their throughput and latency. The core of the experiment is to engineer and evaluate three versions of the same secure web app benchmark and Section 3.1 covers the design for comparing actors, exceptions and futures.

Response time is important when doing authentication and authorisation, as it directly corresponds to how long a user is waiting. The two key metrics for performance speed are throughput - how many requests are being served for a given unit of time and mean latency - how long it takes for a user to receive a response. This experiment measures throughput and latency between the three reliability mechanisms across authentication and authorisation, successful and unsuccessful requests, and a different number of concurrently open connections. The traffic required by the experiment is sent to the server by the load-testing tool wrk.

The three reliability mechanisms for authentication and authorisation introduce a different number of new states in the program. Fewer states means less programming complexity, less attack surface and easier reasoning about software correctness. When there are more states, introduced by the reliability technology, programmers need to make sure that all of these states are secured against potential attackers. This is why we are evaluating the programmability of the three reliability mechanisms as the number of new states introduced.

Secure Web App Benchmark The next achievement is the design and engineering of a secure web service benchmark, implementing the three reliability technologies (Section 3.2).

The service is modelled after a web application, supposed to serve a large number of users where every action needs to be authenticated and authorised. The web server is implemented in Scala, using the Akka-Http framework. Actors, exceptions and futures have two endpoints each - one for authentication and one for authorisation. The three models call the same underlying authentication and authorisation functions. These calls return an error on an unsuccessful operation. The exception model then uses try-catch to handle this error. With actors, the worker is left to crash while the supervisor takes care of the recovery logic. In futures, two callbacks are attached that handle both the successful and the unsuccessful outcome. The service is designed as a monolithic system with the performance benchmarking in mind. Having the same architecture for actors, exceptions and futures will allow us to find differences in performance of the different reliability mechanisms while limiting the impact of potential side effects.

Reliability Mechanisms Evaluation Actors, exceptions and futures are evaluated for authentication and authorisation performance and programmability using the web service benchmark (Chapter 4).

Performance

The three reliability mechanisms are compared and evaluated for their throughput and latency in different conditions. A series of benchmarks are run to measure their performance across different levels of simultaneous open connections.

The first experiment deals with separately sending only successful and only unsuccessful requests, measuring the best performing model for each type, as well as the performance degradation between the two request types. The results of this benchmark are:

- For only successful requests, exceptions have the best throughput across all levels of concurrently open connections
- Actors have the best throughput for unsuccessful requests at 50 connections, and futures - at 3200 connections
- Throughput degradation for unsuccessful requests compared to successful requests is the lowest for actors at 50 connections but the highest at 3200 connections
- Mean latency is lowest for exceptions at 50 concurrent requests for both types

- Unsuccessful requests have low mean latency but significantly reduce the throughput for all three models

The second experiment sends mixed successful and unsuccessful requests, as the latter account for about 11% of the total number. This aims to measure the performance of the three models in a real-world setting, and some key results are as follows:

- The three reliability mechanisms follow similar performance patterns across the different levels of concurrently open connections
- Exceptions have the highest throughput for authentication and authorisation
- Compared to exceptions, actors show a throughput decrease across both security mechanisms of 45.36%, and futures - 17%
- Exceptions have the lowest mean latency
- For all three models, 75% of the responses are received in time less than the mean latency

Finally, the last experiment compares throughput and latency across authentication and authorisation. The results of this experiment show that there is no significant difference in performance for the two different security mechanisms.

Programmability

As a measure of program complexity, we record the number of new states that exceptions, actors and futures each introduce to a program. This is important as fewer states make it easier to reason about a system and its correctness, reducing programmer effort. Furthermore, the fewer the states, the fewer the potential attacking vectors that malicious agents can use, trying to forge a successful authentication or authorisation. The results of the state modelling experiment show that actors have the fewest programming states with 3, futures have 4 and exceptions - 6.

Comparing Benchmarking Tools The final achievement is a short survey comparing two tools for load testing web services - Apache Bench and wrk (Appendix A). It discusses their advantages and disadvantages that were discovered during the project.

Conclusions Receiving mostly unsuccessful requests can severely degrade the performance of the web service. Authentication and authorisation follow the same performance patterns with no major difference between the two. Exceptions have the best throughput and latency of the three models. However, they introduce the most states in the program. Actors are on the other end of the spectrum with the worst throughput and latency but the fewest introduced new states. Futures are in the middle in both categories. In general, there is a clear inverse relationship between performance and programmability.

2 Background

This chapter surveys related work. Section 2.1 introduces the reliability stack and application-level reliability. Section 2.2 introduces the let-it-crash reliability philosophy. Section 2.3 covers cybersecurity, and specifically authentication and authorisation.

2.1 Application-Level Reliability

2.1.1 Reliability Stack

Reliability in software systems has different definitions. In the official ANSI glossary, it is defined as the ability of a system to perform its required functions under stated conditions for a specified period of time (IEEE 1990). In more practical terms, reliability and availability are considered a function of mean time to failure and mean time to repair(Beyer et al. 2016; Baron Schwartz 2015). In the context of this paper, we will be looking at reliability as a measurement of the ability of a system to correct errors and recover from bad states.

In software systems different levels of error correction are present. These levels, based on the layer of abstraction, can be represented as a stack. Every level in the stack contains a component that tries to correct the error on its own. Should it fail to do so, the problem is pushed up the stack. Table 2.1 shows an example representation of the reliability stack.

Table 2.1: *Reliability stack*

People, e.g. Human Error Correction
Application, e.g. Exceptions, Actors, Futures
Network, e.g. IP checksums
Operating System, e.g. page faults, marking bad memory
Hardware, e.g. parity bits

The reliability stack does not have a strict representation. It can be broken down into a variable number of levels. For example, hardware, operating system and network system can be all grouped together into one low-level layer. Alternatively, the network level can be broken down into all the different layers of the OSI model (Briscoe 2000). The part of the reliability stack that we will be exploring in this section is the application level. This is where programming languages and their different error handling mechanisms live. Common mechanisms for application-level error handling are exceptions, futures and actors, and we outline these mechanisms next.

2.1.2 Exceptions

Exceptions in programming languages are events caused by abnormal behaviour. Examples of such events are trying to open a file that does not exist or dividing by zero. It is then up to the developer to write code that handles the possible exceptions. This is usually done by a language provided try/catch construct. If an exception is left uncaught, the program will crash. Some languages have more than one type of exception. For instance, in Java, there are two types of exceptions - checked and unchecked. Checked exceptions must be explicitly handled and this is enforced by the compiler. Handling them is achieved with a try/catch or by adding the exception to the method signature. Unchecked exceptions, on the other hand, are not checked by the compiler and handling them is not enforced. The built-in Java classes Error and RuntimeException are the only ones that fall in this category.

Exceptions have faced substantial criticism, because, from their name, they are supposed to denote very rare or unexpected events. However, in modern programming languages, they are often used to deal with expected and normal events (Siedersleben 2006). The creators of Go cited exceptions creating convoluted code as the reason why they do not exist in the language (The Go Programming Language 2018). Exceptions also provide developers with a mechanism of breaking encapsulation, one of the fundamental principles of Object-Oriented Programming (Dony et al. 2006). Moreover, many software engineers tend to abuse exceptions, wrongfully using them as a control-flow structure (Weimer and Necula 2008). Even in the cases where it is used properly, exception handling adds branching in multiple places in the code. This increases the number of states in the program. Not only does this make it more difficult to reason about the logic in the software, but it also creates more surface for cyber attacks. When the number of states is greater, more attacking vectors that can be abused by attackers are present. We will be exploring what the effects on cybersecurity are in applications, where exceptions can be thrown in multiple states in the program.

2.1.3 Futures

Futures are a mechanism to return values from asynchronous method calls. In different programming languages, they are also known as promises, delegates and others (Liskov et al. 1988). Originally developed as a way to decouple a value from its computation mechanism in functional programming languages, futures have seen a rise recently in the development of responsive user interfaces. In distributed systems, they are a way to minimise the communication overhead, only accessing other hosts when the computation has been finished.

Futures usually provide hooks for attaching a callback that is to be executed when the asynchronous computation has finished. In Javascript, two different callbacks can be registered on a Promise - one for success and one for failure (Mozilla Developer Network 2018). This makes it easy to restart the computation if something goes wrong, ensuring the error is corrected. Assigning timeouts to each future is another efficient and explicit way of keeping an asynchronous process reliable.

Through futures, both the normal and the abnormal outcome can be kept isolated in a separate process. The failure callback handles exceptions of all types. This reduces the number of branches and redundant logic in the program. Furthermore, the same callback can be used for different computations, thus abstracting the error correction process. This is why we are interested in application level reliability via futures.

2.2 Let it Crash

2.2.1 Concept

The let-it-crash philosophy was first proposed by Armstrong (2003). He was working in the Ericsson Computer Science Lab where there was a need for a reliable and fault-tolerant telecommunication switching system. The key requirements were that it had to run for a long period of time and be able to deal with multiple concurrent activities, distributed across multiple hosts. Acknowledging that no process can be 100% reliable because hardware failures exist, Armstrong argued if the process or the host failed, it would not be able to recover on its own. Because of that, he proposed at least two communicating hosts with a process running on each. If a failure occurred in one of the processes, the other was responsible for handling it appropriately. Handling it in that particular example meant restarting the process from a clean state. The expected behaviour of the failing process was simply to crash, hence the name. The idea is also scalable to multiple processes on a single host or multiple processes on multiple hosts. Processes can look after each other and handle failures no matter whether on the same physical machine or not.

The work of Joe Armstrong and others led to the development of Erlang and the Open Telecom Platform (OTP) (Erlang 2018). It suggested a supervision model for concurrent processes. According to this model, the processes are represented as a tree graph in which leaf nodes are "worker" processes, while all other processes are "supervisors". If a supervisor detects a fault in one of its "supervisees" it can restart the process. Alternatively, if the other children of this supervisor were dependent on the failing process, all processes can be started from a fresh state. If it fails to handle the problem, the supervisor itself can crash and propagate the fault to the supervisor at the root of the tree. Utilising the let-it-crash philosophy of supervisors and workers increases the robustness and redundancy of a system. It reduces the need for defensive programming at every step of the code, where covering every error case might prove to be difficult. Reliability is ingrained in the architecture, rather than deep in the computation implementation. Although it is named let-it-crash, the concept essentially means writing robust, self-healing software.

In recent years, ideas drawn from the let-it-crash concept have been heavily featured in micro-service architectures. Container orchestrators, such as Kubernetes (Kubernetes 2019) can automatically restart a crashed service, similar to the supervisor restarting a worker process. Services, such as Netflix's Chaos Monkey (Netlifx 2018), are specifically designed to kill groups of processes, running in production. Libraries like Akka for Scala implement Erlang-style supervision for their actor processes. Similar to Armstrong's idea from 40 years ago, occasional faults are unavoidable. Therefore software developers should write robust, fault-tolerant applications, that can continue from where they left off. The goal in these recently popularised services and paradigms is once again increased reliability of the system and self-healing software.

2.2.2 Actors for Concurrent Programming

The actor model was first suggested by Hewitt et al. (1973). The reasoning behind it was the prospect of highly parallel computing machines communicating via a high-performance network. Erlang and its processes, developed by Joe Armstrong, are an implementation of the actor model. They build on top of the initial ideas suggested by Hewitt, through the concepts of supervision and fault-tolerance.

The actor model presents an abstraction in which actors are the universal primitive computation unit. Similar to object-oriented paradigms where everything is an object, in actor-based models everything is an actor.

Every actor has its own local state and resources are not shared. This provides a different type of concurrency, compared to, for instance, the threaded model which is based on the idea of shared memory and mutual exclusion. The distributed memory, used in the actor model, makes the entire system more reliable. If one of the actors fails, the rest are not affected, because of the resource isolation. Furthermore, using distributed memory gets rid of common shared-memory caveats, such as race conditions and deadlocks.

Actors communicate with each other via message passing. Based on the received message, an actor can take different actions, e.g. send a message, start a computation or spawn another actor. These actions can be carried out asynchronously and in parallel. An actor can send messages only to actors whose address it knows. Actors provide no guarantees of message order or arrival. Every actor can run on a single or on multiple threads.

2.2.3 Actor Languages & Frameworks

Erlang. Erlang is a concurrent, functional programming language. It runs on the Erlang virtual machine, also known as BEAM VM. It was the first popular language to implement the actor model. Actors in Erlang are called *processes*. Unlike OS-level processes, Erlang processes are lightweight and have a small memory footprint. These processes can be linked to each other and can follow the "let-it-crash" philosophy. Erlang comes with the OTP(Open Transport Protocol) which provides a set of libraries for debugging, interfacing with third-party service (e.g. databases) and handling releases.

The original use case for Erlang was in telecommunications and developing telephone switching systems, where its adoption led to great success (Armstrong 2007). In recent years, it has been the driving force behind one of the biggest instant messaging platforms, Whatsapp (Reed 2012).

Elixir. Elixir is a language influenced by Erlang and Ruby. It also runs in the BEAM VM and has seamless API integration with Erlang. It has been recently recognised by Joe Armstrong 2013 as a great language building on top of Erlang, fixing some of its underlying issues. Elixir uses the same lightweight processes for concurrency as Erlang. It is a popular language with a strong community following. At the time of this writing, its Github repository has nearly 14 thousand stars and more than 2 thousand forks.

Scala. Scala is a high-level language combining elements from object-oriented and functional programming. It runs in the Java Virtual Machine and provides language interoperability with Java. Scala used to provide its own implementation of the actor model, fully based on the one present in Erlang. The actors were lightweight, with small memory footprint and local memory. In version 2.10, Scala actors were deprecated in favour of the Akka implementation (Philipp Haller and Stephen Tu 2013).

Akka is one of the most popular actor model frameworks. It builds on top of Scala's functional programming capabilities and adds first-class actor support. Akka fully embraces the supervisor-worker paradigm and every actor needs to be assigned a supervisor. Thus the let-it-crash philosophy is enforced by default. Every actor can be registered in the `ActorRegistry`, allowing it to be looked up by other entities. On a lower level, Akka uses the Akka remote protocol for message passing. This allows actors on local and remote hosts to be treated equally.

Others. vert.x (Eclipse Vert.x 2019) is a framework for a wide selection of programming languages - namely Java, JavaScript, Groovy, Ruby, Scala, Kotlin, Ceylon and Clojure. It runs in the JVM and provides a standard actor model implementation.

Akka.NET (Akka.NET 2019) is an Akka port for the .NET family of languages. It is an open-source project, powered by a strong community behind it. It has all the features and advantages of the original Akka framework.

Protoactor (Proto.Actor 2019) is a cross-platform actor framework for .NET, Go and Java. It uses protobuf and gRPC and makes no difference between local and remote actors. It also provides supervising by default.

2.3 Security

Security is an important part of any modern application. People give software access to their birthdays, credit card numbers, photos, etc. This raises the need for the system to ensure that this private data will not go in the hands of other users or attackers. Failing to do so can be unethical and illegal (Union 2016).

The two main security mechanisms for protecting users and their information are authentication and authorisation. When the user interacts with the software these verifications are typically done more than once. For example, data access actions require the application to authorise them first. Consequentially, a lot of parts of the source code will need to perform authentication and/or authorisation for the logic to be executed. This can lead to a lot of boilerplate with a large number of states in the program.

When an authentication or authorisation fails, the service enters an error state. Application-level reliability mechanisms are then responsible for restoring the system back to a normal state and responding properly to the user. These mechanisms need to be secure enough to prevent potential malicious agents from interfering in the system. Each additional state that the reliability technology introduces in a program can potentially lead to more attack surface that can be abused. Furthermore, the mechanisms need to keep up with the high performance and low latency users tend to expect. An optimal reliability technology will have high performance and very few additional programming states.

2.3.1 Authentication

Authentication is the process of verifying the identity of a user, process, or device (Kissel 2013). This is usually done as a prerequisite for granting access to resources in an information system. One of the most common authentication mechanisms is asking a user to provide a name and a password. Recently, in an effort to increase security and reduce identity theft in the online world, methods, such as multi-factor authentication, have emerged. In 2-factor authentication, after supplying the correct name and password combination, the user is prompted for a unique code. This one-time use code is sent to a trusted device or phone number. A common practice in web applications is to create an identity cookie after authenticating. The user is then automatically authenticated until the cookie expires.

Authenticating processes and devices is usually done with a shared secret, e.g. a unique key or a key and password combination. The application can use this secret to prove its identity upon initial connection or every time it requests data from an API. Authentication is a standard problem in cybersecurity and although implementations might change, it will always be present in some shape or form.

2.3.2 Authorisation

Authorisation is the process of granting access privileges to a user, program, or process (Kissel 2013). In information systems, this is usually the act of allowing an authenticated entity to read or write particular data. Properly limiting resource access for different users and programs is important and errors in the authorisation logic can lead to data leaks with huge monetary and political consequences.

Authorisation is present in all levels of the hardware and software stack, starting from the CPU where there are different logic units that can be accessed only by some programs. In network communications, not all packets are supposed to be opened by everyone. Authentication and authorisation in the form of encrypting and identity checking are used in the network and transport layers. Operating systems have users and groups with different permission levels. What we are interested in this paper is application level authorisation. In web services, authorisation is usually a process that happens directly after authentication. In a simple web server scenario, when an entity accesses an endpoint, their identity has to be established and then whether they have the permission to use that particular resource.

3 | Experiment and Benchmark Design and Implementation

This chapter starts by presenting the philosophy and design of the experiment to evaluate the alternative web application security technologies (Section 3.1). The experiment evaluates performance, i.e. throughput and latency, and programmability, i.e. number of program states. The core of the experiment is the design and implementation of a common secure web application kernel benchmark in each of the reliability technologies, i.e. with actors, exceptions and futures (Section 3.2).

3.1 Performance and Programmability Experiment Design

The goal of the experiment is to compare the three reliability mechanisms for engineering secure web apps, focusing on performance and programmability. The basis of the experiment is a common secure web application built using actors, exceptions and futures, as described in Section 3.2.

Performance is important for measuring the end-user implications of each reliability mechanism, e.g. an increase in latency caused by failures. Subsection 3.1.1 covers the experiment part focused on the performance implications, in terms of throughput and mean latency.

Programmability is a metric, concerned with programmer effort and general program complexity. A higher level of programmability means it is easier for engineers to reason about a program's correctness. This is done through validating every state the program can possibly enter. The number of states in a program and their correctness has major security implications. Fewer states in a system means less attack surface for potential attackers. Furthermore, the fewer the states, the less the programming effort required to ensure they are all secure. Subsection 3.1.2 explores the experiment part focused on programmability.

3.1.1 Performance

Metrics The two main performance features that are measured in a networked software system are throughput and mean latency. Throughput is a metric of how many requests are being served for a given unit of time. Mean latency is how long it takes on average for a given request to receive a response from the server. These two metrics together can give a good representation of how well a service is performing. Poor results in either can mean a bottleneck in the server architecture or the implementation. Engineers generally aim for high throughput and low latency services, but one does not always guarantee the other. For example, inappropriate queuing mechanisms can focus primarily on newly received requests, responding immediately to them, while keeping old connections unprocessed for a long time. This keeps the throughput high but some users will suffer from very high latency.

The experiment on comparing the three reliability mechanisms is focused on measuring their throughput and mean latency in different scenarios. As both of these metrics can be influenced by multiple conditions, we vary the different parameters and explore whether particular reliability models deal better than others in some circumstances. These parameters are discussed next.

Concurrent Connections In order to measure the throughput and latency in different conditions, the service and the reliability mechanisms are stress tested. One way this can be achieved is by simply increasing the number of requests that are being sent. However, this does not necessarily put any additional pressure on the server, as even with an increased amount, these requests are being sent sequentially. This is easy to deal with by the service in the general case. Instead, the number of concurrent connections that are open at a given time can be increased. This way requests are being sent to the server in parallel. By consistently increasing the number of open connections, it can be reasonably expected that the throughput will be increasing, as long as the service is capable of handling all requests. There can be a point, though, after which the throughput plateaus and even starts falling down. It is reasonable to expect that mean latency goes up, as the number of concurrent connections is being increased.

The number of concurrently open connections that this experiment explores starts at 50. Every subsequent run doubles that amount until a ceiling of 3200 is reached.

Request Types The second key parameter that can be changed is the requests that are being sent to the server. The reliability mechanisms act differently, depending on whether the authentication or authorisation was successful or not. Sending a request, supposed to fail, can enter the system in an abnormal state. The ability of the service to handle these abnormal states is directly related to the choice of the reliability mechanism. The experiment explores how actors, exceptions and futures behave, based on the different requests that they are dealing with. It is likely that one of them has better throughput than the others for successful requests but worse throughput for unsuccessful ones. Additionally, mean latency can be influenced by the reliability mechanism and how long it takes for the system to return to its normal state.

The experiment analyses several different scenarios. The first two scenarios deal with handling only successful and only unsuccessful requests respectively. Comparisons between the two explore the reliability mechanisms performance in different settings. The final scenario, with regards to the request type, looks at a realistic split between the two requests. According to Mare et al. (2016), users normally fail about 11% of their authentications in everyday use. The reasons behind this can be forgotten passwords, wrong usernames, etc. This realistic setting is simulated by sending an 89/11% split of successful and unsuccessful requests in order to see which reliability mechanism performs the best in a real-world scenario.

Security Mechanisms The last parameter that is explored is the difference in performance across authentication and authorisation. Both of these security mechanisms play an important part in modern web services, so it is important to investigate whether the choice of a reliability technology can influence their performance in different ways. However, considering that the difference between authentication and authorisation is mostly in the final function call that is being executed by the server, it is expected that the two will follow similar patterns. As they have the same infrastructure and reliability mechanisms are overseeing both actions, a performance difference will most likely mean a bottleneck in the authentication or authorisation function itself.

This relationship is explored by running the experiment with the same exhaustive list of scenarios, mentioned above, for both security mechanisms. This helps determine whether they follow similar patterns and if not, whether this might potentially be because of the reliability technology.

Traffic Generation Tool The stress testing tool of choice for generating traffic is wrk (wrk 2019). It is a fast and lightweight, multithreaded application for load testing servers. Wrk uses event notification systems to ensure the quick performance and proper synchronisation, no matter the number of threads, requested by the user. It provides options for specifying the number of concurrently open connections and either the total number of requests or the time for the benchmark to run. Furthermore, it exposes a Lua scripting API for dynamically changing the requests during runtime with no performance penalties. This allows to easily achieve the desired split of successful and unsuccessful requests. The API also enables exporting the results in a format that eases the evaluation process.

In the early stages of the project, another tool was considered for the execution of the experiment - Apache Bench (Apache Bench 2019). However, due to its single-threaded design and limited implementation, the results obtained with it were poor and unreliable. Because of that Apache Bench was later dropped in favour of wrk. A thorough comparison of the two traffic generation tools is covered in Appendix A. The scripts used for running the Apache Bench experiments are in the `ab_scripts` directory in the repository.

3.1.2 Programmability

While performance is a metric, concerned with how well the system runs and how satisfied users are with the overall experience, programmability deals with programmer effort and programming complexity of the three reliability mechanisms. This is important not only because it directly relates to the engineering time necessary, but also because of ensuring program correctness. Developers need to be able to reason about why their software is robust and correct and this is especially important when it comes to cybersecurity. Better programmability can mean safer software. The different reliability mechanisms are based on different concepts and as such, they carry different levels of programming complexity, associated with them. It is assumed that all three of them are equally familiar to an engineering team, making the choice between the three models.

There are several ways of measuring the programmability of the different reliability mechanisms. One idea is to simply count the number of lines that each of them brings to the program. However, this is a rather naive metric, as it is highly dependent on coding standards and conventions, as well as the code context. Furthermore, less or more lines of code do not guarantee any functionality or correctness in the software. Alternatively, the proposed metric for programmability in this project is to use state modelling. The number of new states that each of the reliability mechanisms introduces to the program relates directly to the available attack surface that can be targeted by malicious agents. Fewer states implicitly means less potential attacking vectors. With lines of code, there is no such guarantee, as often multiple lines of code are responsible for the same piece of functionality. Finally, ensuring the correctness and security of every state is a reliable way for guaranteeing system soundness. Thus, fewer states means less programming effort for writing secure and robust software, hence a higher level of programmability.

As a part of the experiment, the states introduced by each of the three reliability mechanisms are modelled and compared in terms of the number of new states introduced in the program.

3.2 Common Secure Web App Benchmark

3.2.1 Design

The web service is designed as a basis for the experiments outlined in the previous section. A common code base is used, excepting that authentication and authorisation failures are handled separately by actors, exceptions and futures in the three versions. The three reliability mechanisms are evaluated for their throughput and latency. This means that the other parts of the system need to be consistent across actors, exceptions and futures. In this section, some of the design decisions that were made in the project and the rationale behind them are discussed. Figure 3.1 shows the finalised secure web service design.

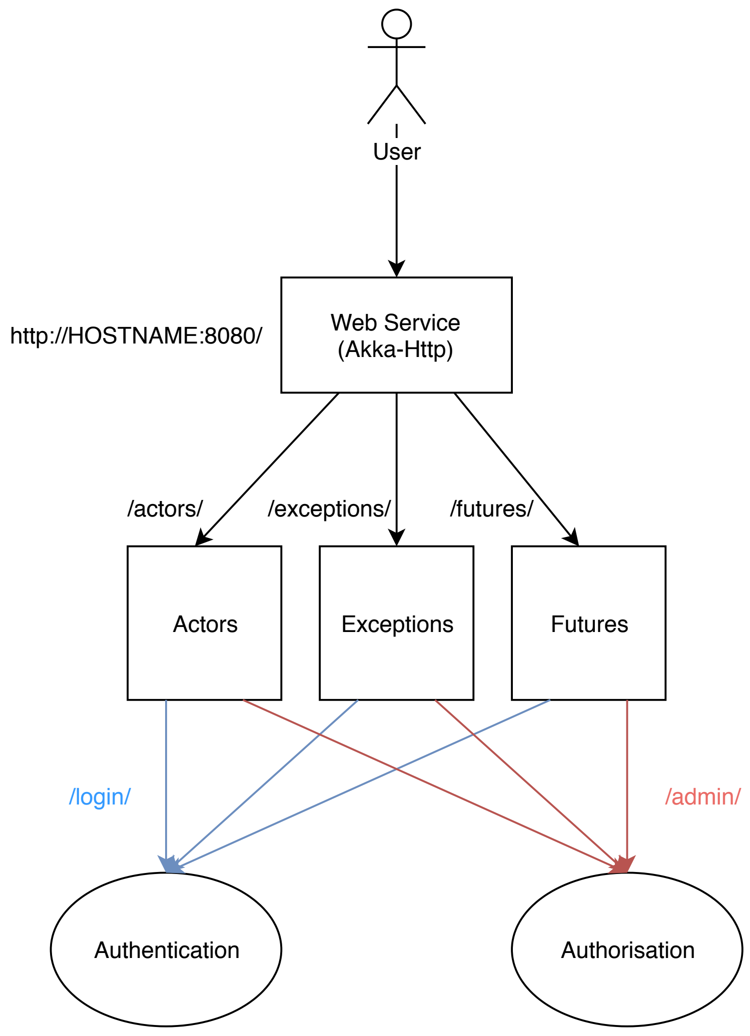

Figure 3.1: *Common Secure Web Application Design*

Programming Language and Framework The choice of a programming language for the secure web app benchmark is based on industrial relevancy and experiment suitability. Scala is one of the most popular server languages (W3Techs 2019). It has recently been used as the driving force behind Big Data applications like Kafka, Flink and Spark (Miller et al. 2016). Scala is also used for the server-side infrastructure of tech-giants, such as LinkedIn (bagwell 2010), Twitter (Marius Eriksen 2012) and Ebay (Deepak Vasthimal 2016; Blesson Paul 2016). Furthermore, it offers mechanisms for implementing each of the three reliability models. Moreover, Scala's rich variety of features can make it difficult for engineers to make an informed design decision on which feature set is the best fit for their application. This project aims to answer this requirement when it comes to authentication and authorisation.

Several different web service frameworks were considered for the development of the benchmark. Based on research, the shortlisted options were Scala Play (Scala Play 2019), Scalatra (Scalatra 2019) and Akka-Http (Akka HTTP 2019). Ultimately, the framework of choice was Akka-Http, because it is lightweight and provides only the necessary functionality. For example, Scala Play implements the Model-View-Controller pattern and generates many files and abstractions that are not necessary for the project's realisation. Akka-Http allows us to have a simple web server that accepts a request and routes it to a particular function, or in our case - reliability model, that handles everything after that.

Routing After the web service receives a request, it redirects it to one of the three reliability mechanisms, depending on the endpoint that has been hit. This routing follows a consistent pattern where the first part of the path after the hostname is the requested reliability mechanism, and the second part is the requested security function - authentication or authorisation. Actors, exceptions and futures have two endpoints each. Although it would not make any difference in terms of performance, an alternative design would be to have the first part of the path specify the security mechanism, and the second one - the reliability model. However, this was discarded in favour of the first option, because the key part for this project is whether this request is going to actors, exceptions or futures. This design would also make the eventual implementation more straightforward and clear.

Reliability Mechanisms The reliability models are designed in an idiomatic way and according to their philosophies. The actor model has a supervisor and a worker actors. If the worker fails to authenticate or authorise the request, it just crashes and it is up to the supervisor to restore the system back to its normal state. In the exception model, there are try-catch clauses that handle the successful and unsuccessful outcomes. Handling of different exceptions and finally-clauses are omitted for the sake of brevity. With futures, after calling the authentication or authorisation functions, a callback is attached to handle both the successful and the unsuccessful case. Each of the three reliability mechanisms has a controller that is used to call the authentication or authorisation function. This controller is, in essence, the public interface. The implementation details of the actor, exception and future models are covered in Sections 3.2.2, 3.2.3 and 3.2.4 respectively.

Authentication and Authorisation The authentication and authorisation functions are designed for simplicity. A decision was made against using a database, in order to avoid IO performance bottlenecks. Instead, all the user data is stored in memory which allows the secure web app benchmark to focus its throughput and latency measurements on the reliability mechanisms.

The authentication function emulates a user trying to log into the web application. It receives a username and password pair, in the form of Basic Auth (Reschke 2015). This is then checked against a map that contains the registered users and their passwords. No password hashing or encrypting is performed, as this is not the intended part of the research for the project. In a real-world production environment, the passwords should be stored in a database and not in plain-text format. On successful authentication, the function completes without errors and the reliability mechanism returns an HTTP status code 200. On failure, an exception is thrown and after being handled by actors, exceptions or futures, a response with HTTP status code 401 Unauthorised is returned.

The authorisation function implicitly assumes a previous authentication that has been done. In this previous step, the user has been issued a cookie which represents their identity, in order to avoid unnecessary username-password combination checks. The authorisation function simulates a user trying to access the admin panel. The cookie passed in the request is mapped to a user and their permissions are then checked. If the user is a part of the admin group, the function completes successfully and the reliability mechanism returns an HTTP status code 200. On failure, an exception is thrown and after being handled by the respective model, a response with HTTP status code 403 Forbidden is returned.

3.2.2 Actors Implementation

Three different actor types are defined - `Supervisor`, `AuthenticationWorker` and `AuthorisationWorker`. The supervisor is shared between the two different types of workers. Actors communicate with each other via messages. The three types of messages specified are `Authenticate`, `Authorize` and `CompleteFailed`. The first two have an additional payload which is the credentials and the cookie respectively.

When initialising the actor system, a fixed number of supervisor actors is spawned. This is done because having only one supervisor for a potentially large number of workers can be a performance bottleneck. However, automatically scaling and reducing the active supervisors is a convoluted task, that requires a complex implementation. As the benchmark focuses on simplicity and keeping things as basic as possible, a decision is made to go with a static number of 10 supervising actors. The requests are distributed between the supervisors in a round-robin fashion. The selected number of supervising actors allows for a respectable level of fan-out even in high-load situations. Furthermore, the assumption is that more often than not workers will be able to deal with the request on their own, i.e. they will not crash.

After the supervisor receives the message to authenticate or authorise, it spawns the respective worker actor and forwards the message to it. If the operation executes successfully, the worker completes the request and dies. Should it fail to authenticate or authorise the user, though, it crashes. The supervisor has a one for one supervising strategy. This means that even if there are multiple children in the supervising hierarchy, it only takes action towards the child that has crashed. The strategy in use makes the supervisor restart the crashed actor and immediately send it a `CompleteFailed` message. The worker completes the request with the appropriate error response (401 or 403) and dies.

Listing 3.1 shows a code snippet for authentication with actors. It presents the `Supervisor` and `AuthenticationWorker` actor classes. The supervisor overrides the default strategy with a `OneForOneStrategy` where the worker is restarted and sent a message to complete immediately with a failed response. Both actor classes specify the message types to pattern match on. Finally, the constructor for the `AuthenticationWorker` takes two parameters - authentication credentials, i.e. username and password, and a function pointer for completing the request.

```
class Supervisor() extends Actor {
  override val supervisorStrategy: OneForOneStrategy =
    OneForOneStrategy() {
      case _: Exception =>
        sender() ! CompleteFailed
        Restart
    }

  override def receive: Receive = {
    case Authenticate(credentials, complete) =>
      val props = Props(new AuthenticationWorker(credentials = credentials,
          complete))
      val worker = context.actorOf(props)
      worker ! Authenticate
    }
}

class AuthenticationWorker(credentials: Option[HttpCredentials], complete:
    StatusCode => Unit ) extends Actor {
  override def receive: Receive = {
    case Authenticate =>
      authenticate(Credentials(credentials))
    case CompleteFailed =>
      complete(StatusCodes.Unauthorized)
      context stop self
  }

  def authenticate(credentials: Credentials): Unit = {
    Common.authenticate(credentials)
    complete(StatusCodes.OK)
    context stop self
  }
}
```

Listing 3.1: Authentication with actors

3.2.3 Exceptions Implementation

The implementation of the exception model uses the same underlying authenticate or authorise function as the actor model. They are called in the try statement. If that succeeds, the successful status code is returned. The catch handles every possible exception that might occur and completes with the appropriate HTTP error code. Most languages, including Scala, provide a finally-block functionality which executes after both the try and the catch. However, for the sake of simplicity, this clause is omitted in the secure web app benchmark implementation.

Listing 3.2 shows a code snippet for authentication with the exceptions reliability mechanism. The `authenticate` function takes two parameters - the username and password credentials, and a function reference for responding to the request. If the authentication fails, the catch block handles all possible exceptions and returns a failed response.

```
def authenticate(credentials: Option[HttpCredentials], complete: StatusCode =>
    Unit): Unit = {
  try {
    Common.authenticate(Credentials(credentials))
    complete(StatusCodes.OK)
  } catch {
    case _: Exception => complete(StatusCodes.Unauthorized)
  }
}
```

Listing 3.2: Authentication with exceptions

3.2.4 Futures Implementation

Futures in Scala require an `onComplete` callback to be attached to each function that returns a Future object. The callback needs to handle both the successful and failing case. The authentication and authorisation methods in the `FutureController` use the same underlying functions as actors and exceptions. `onComplete` is then attached, returning a successful HTTP code if no errors occurred, and Unauthorized or Forbidden if they did.

Listing 3.3 shows a code snippet for authentication with the future model. The `authenticate` function takes the same parameters, like the one for exceptions, but returns a different type. The `Future` type is resolved through the `onComplete` function, which resolves both the successful and the unsuccessful case.

```
def authenticate(credentials: Option[HttpCredentials], complete: StatusCode =>
    Unit): Unit = Future {
   Common.authenticate(Credentials(credentials))
}.onComplete {
   case Success(_) => complete(StatusCodes.OK)
   case Failure(_) => complete(StatusCodes.Unauthorized)
}
```

Listing 3.3: *Authentication with futures*

4 Evaluation

This chapter discusses the results from the designed experiments and their implications. Section 4.1 covers the specific language and framework versions, as well as the hardware configuration on which the secure web application benchmark was run. The results from the different performance experiment scenarios are presented in Section 4.2. The throughput and latency are discussed for each experiment, highlighting the conclusions drawn. Programmability of the three models is analysed in Section 4.3. The number of states in each mechanism is counted and a critical discussion is provided on the key findings.

4.1 Hardware and Software Context

The language and framework versions used are as follows:

- Scala 2.12.7
- Akka 2.5.12
- Akka-Http 10.1.5
- sbt 1.2.6
- JDK 1.8.0

The experiments were run on two nodes of the Glasgow Parallelism Group cluster. One of the nodes was running the secure web app benchmark, while the other - the load testing tool wrk. Each of these nodes has the following characteristics:

- 16 Intel cores (2 * Intel Xeon E5-2640 2GHz)
- 64 GByte RAM
- 10 Gbit Ethernet connection
- Ubuntu 14.04

4.2 Performance Experiments

Each of the experimental scenarios runs for 5 minutes with the specified number of concurrently open connections. This allows for enough time for the Java Virtual Machine garbage collection to trigger and thus prevent skewed results. Furthermore, the Java Virtual Machine is restarted between the different types of experiments. Reported results are the median values of three consecutive benchmark runs.

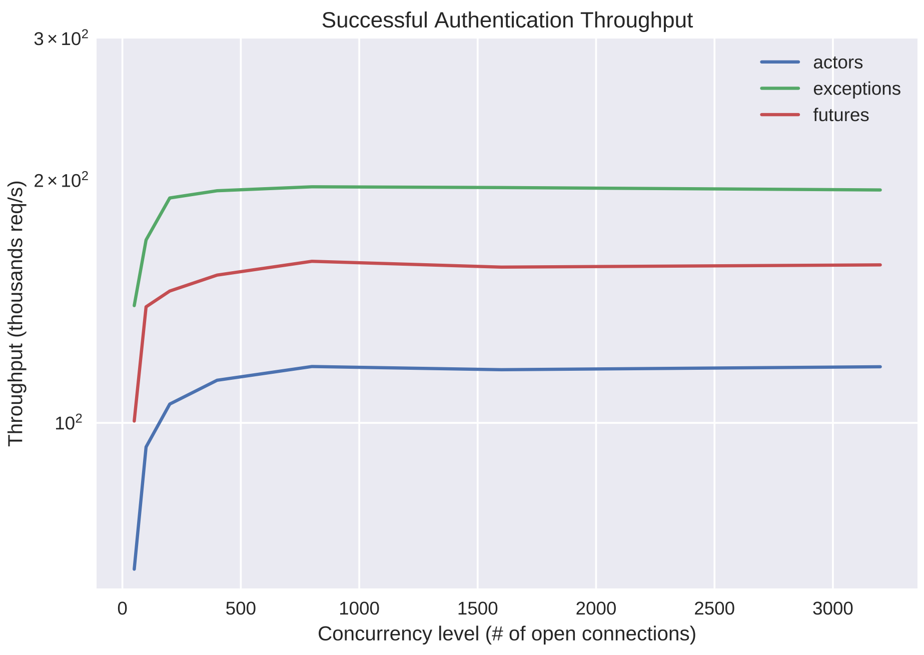

Figure 4.1: Throughput for Authentication with 100% Successful Requests

4.2.1 Successful and Unsuccessful Requests

The first experiments are concerned with the performance implications of the server dealing with only successful requests, and only unsuccessful requests. It measures the throughput and latency for each of the reliability mechanisms when dealing with each type of request. This is done for both authentication and authorisation. The aim of this experiment is to find the performance of which reliability mechanism is the best in both situations.

The results for authorisation follow a similar pattern as those for authentication. Figures, showing the results for authorisation are available in Appendix B. Unless explicitly stated otherwise, the observations made in this subsection hold for both authentication and authorisation.

Throughput Figure 4.1 shows that exceptions have the highest throughput for 100% successful requests across the different levels of concurrently open connections, reaching 196,346 requests per second with 800 connections. In contrast, futures reach just 158,692 requests per second with 800 connections, and actors just 117,477 requests per second with 800 connections.

However, the results are different for 100% unsuccessful requests (Figure 4.2). Actors have the highest throughput at 50, 100, 200 and 400 concurrently open connections – 2,360, 1544, 828 and 517 requests per second respectively, compared to 183, 233, 213 and 241 requests per second for exceptions, and 333, 428, 455 and 495 requests per second for futures. The throughput decreases for actors with an increase in concurrently open connections. This can be attributed to the inability of the limited number of supervisors to handle the large number of crashes that are happening. In contrast, futures reach their highest throughput at 3200 concurrently open connections with 671 requests per second, compared to 123 requests per second for actors and 365 requests per second for exceptions at this concurrency level. The difference in performance between the two request types for the three reliability models is evident with the throughput for unsuccessful requests being approximately 1000 times less than for successful requests.

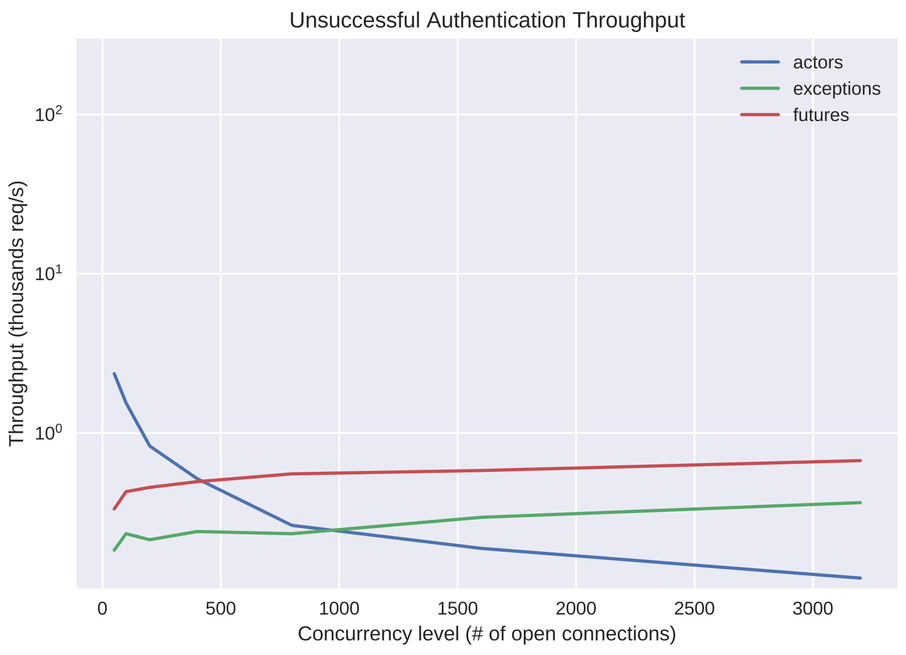

Figure 4.2: *Throughput for Authentication with 100% Unsuccessful Requests*

Table 4.1 explores further the throughput degradation from successful to unsuccessful requests in actors, exceptions and futures. It presents the percentage decrease at each concurrency level for each of the reliability mechanisms for both authentication and authorisation. This analysis shows that actors have the smallest decrease in throughput at 50 concurrently open connections – 96.41% for authentication and 95.98% for authorisation. In contrast, exceptions have 99.87% for authentication and 99.86% for authorisation, and futures have 99.67% for authentication and 99.66% for authorisation. However, at 3200 concurrent connections actors have the largest performance degradation with 99.9% for authentication and authorisation. On the other hand, exceptions and futures have remained consistent with 99.81% and 99.57% for authentication and 99.83% and 99.65% for authorisation, respectively. The overarching degradation pattern further confirms the huge throughput difference between only successful requests and only unsuccessful requests.

Mean Latency Figures 4.3 and 4.4 show that exceptions have the lowest mean latency between the three reliability mechanisms for both 100% successful and 100% unsuccessful requests. All three mechanisms have their best results for 50 concurrent connections. Exceptions record 0.54ms for successful requests and 0.25ms for unsuccessful requests, futures – 1.19ms for successful requests and 0.38ms for unsuccessful requests, and actors – 1.67ms for successful requests and 0.63ms for unsuccessful requests. At 3200 concurrent connections, actors have the worst performance for the successful scenario – 9.19ms, compared to 6.98ms for futures and 5.7ms for exceptions. In contrast, for the unsuccessful scenario futures have the worst performance – 2.04ms, compared to 1.49ms for actors and 1.33ms for exceptions.

It is interesting to note that the mean latency is actually better for unsuccessful requests than for successful ones. The unsuccessful scenario latency is 3-6 times lower than for the successful one. This is unusual considering the throughput of the system has dropped significantly, as already observed in Table 4.1. These observations point that the service handles failing requests

Table 4.1: *Throughput Decrease for Unsuccessful vs. Successful Requests*

Open Connections	Authentication			Authorisation		
	Actors	Exceptions	Futures	Actors	Exceptions	Futures
50	96.41%	99.87%	99.67%	95.98%	99.86%	99.66%
100	98.35%	99.86%	99.69%	98.42%	99.87%	99.7%
200	99.21%	99.89%	99.69%	99.14%	99.88%	99.7%
400	99.54%	99.88%	99.67%	99.59%	99.9%	99.7%
800	99.78%	99.88%	99.65%	99.77%	99.87%	99.71%
1600	99.84%	99.85%	99.63%	99.87%	99.88%	99.65%
3200	99.9%	99.81%	99.57%	99.9%	99.83%	99.65%

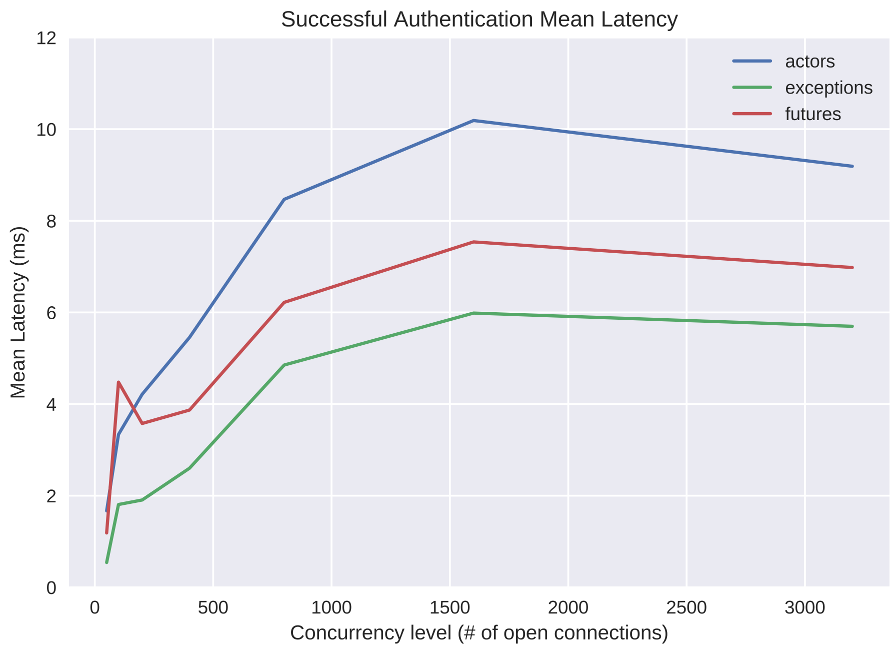

Figure 4.3: *Mean Latency for Authentication with 100% Successful Requests*

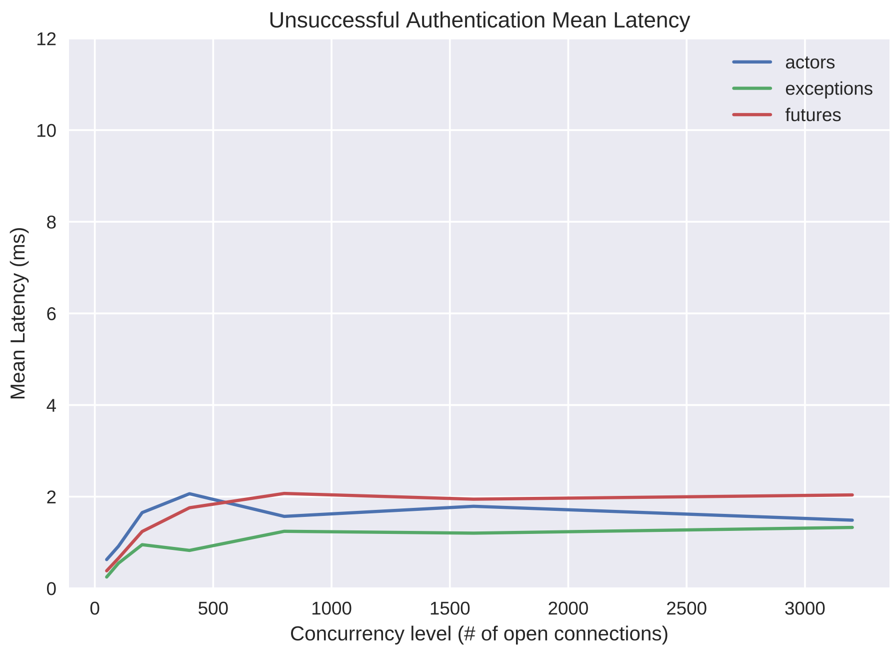

***Figure 4.4:** Mean Latency for Authentication with 100% Unsuccessful Requests*

by focusing on latency. The server tries to respond as quickly as possible to an unauthenticated or unauthorised request but then takes additional time to restore the system to its normal state. This results in low latency and low throughput.

Key Findings

- **Exceptions have the best throughput for only successful requests across all concurrency levels**, peaking at 196,346 requests per second with 800 connections for authentication, compared to 158,692 requests per second for futures and 117,477 requests per second for actors (Figure 4.1)
- **For only unsuccessful requests, actors have the best throughput at 50 connections** – 2,360 requests per second for authentication (compared to 183 requests per second for exceptions and 333 requests per second for futures), **and futures at 3200 connections** – 671 requests per second for authentication (compared to 123 requests per second for actors and 365 requests per second for exceptions) (Figure 4.2)
- **Throughput degradation is the lowest for actors at 50 connections** – 96.41% for authentication, compared to 99.87% for exceptions and 99.67% for futures; however, it is increased to 99.9% at 3200 connections, while exceptions and futures stay consistent at 99.81% and 99.57% respectively (Table 4.1)
- **Mean latency is lowest for exceptions at 50 concurrent requests for both request types** – 0.54ms for successful requests and 0.25ms for unsuccessful requests, compared to actors – 1.67ms for successful requests and 0.63ms for unsuccessful requests, and futures – 1.19ms for successful requests and 0.38ms for unsuccessful requests (Figure 4.3 and 4.4)
- **While unsuccessful requests have low mean latency, they dramatically reduce the throughput for all three models** (Figure 4.1 and 4.2)

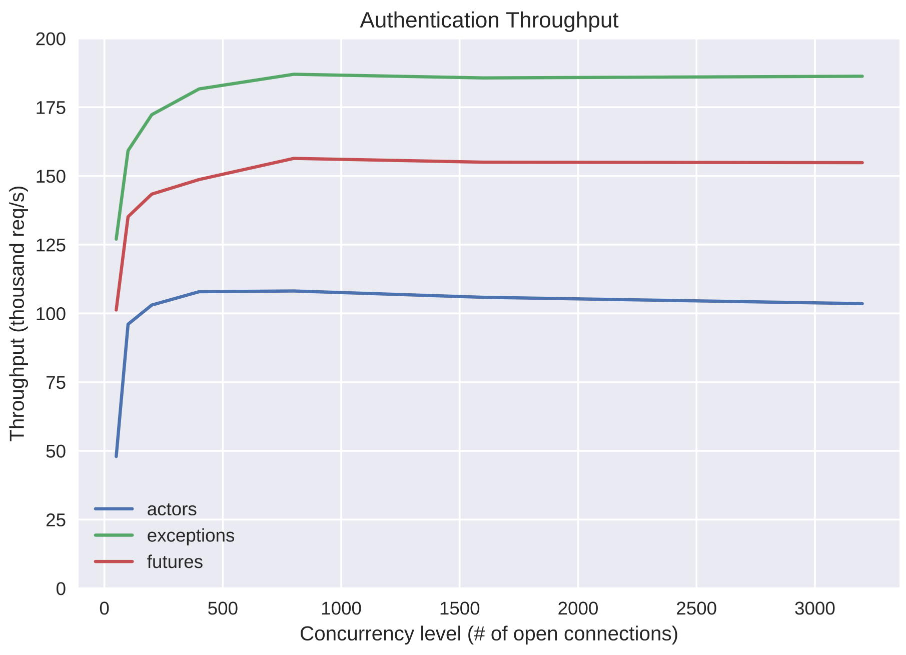

Figure 4.5: Authentication Throughput for a Realistic Request Split

4.2.2 Mixed Requests

Comparing the performance of the three reliability mechanisms should be ideally done in a realistic environment. Servers in production do not deal with exclusively succeeding or failing requests but rather a mixture of both. This experiment measures the throughput and the mean latency of actors, exceptions and futures when dealing with an 89/11% split of successful and unsuccessful requests. We are interested which is the best performing model in this particular scenario and how close its results are to those of the other models. The experiment is carried out for both authentication and authorisation.

Throughput Figure 4.5 and Figure 4.6 show the throughput for a realistic request split for authentication and authorisation respectively. The two graphs show that actors, exceptions and futures follow the same pattern across the different levels of concurrently open connections. Throughput hits its highest point at 800 connections and then it plateaus. Exceptions have the highest number of requests per second for authentication with 186,891 requests per second, followed by futures with 156,379 requests per second and then actors with 108,154 requests per second. For authorisation, the same pattern follows, with exceptions being first with 189,737 requests per second, futures with 156,274 requests per second and actors with 104,998 requests per second. Table 4.2 presents the absolute values of exceptions throughput and the percentage decrease in the other models. It shows that across both security mechanisms and all levels of concurrently open connections actors have an average performance decrease of 45.36% compared to exceptions, and futures – 17%.

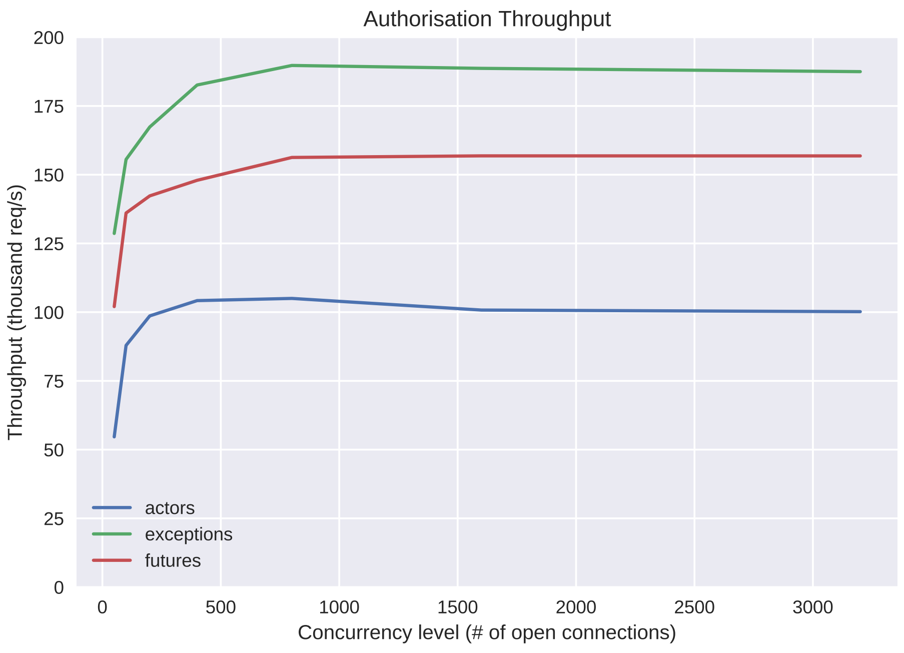

Figure 4.6: Authorisation Throughput for a Realistic Request Split

Table 4.2: Realistic Split Throughput for Exceptions and Percentage Decrease for Actors and Exceptions

Open Connections	Authentication			Authorisation		
	Exceptions	Actors	Futures	Exceptions	Actors	Futures
50	127k req/s	62.25%	20.27%	128.6k req/s	57.52%	20.69%
100	159.2k req/s	39.66%	15.1%	155.6k req/s	43.49%	12.53%
200	172.3k req/s	40.19%	16.77%	167.3k req/s	41.06%	14.97%
400	181.6k req/s	40.6%	18.14%	182.6k req/s	42.94%	18.98%
800	187k req/s	42.16%	16.37%	189.7k req/s	44.66%	17.64%
1600	185.6k req/s	42.98%	16.51%	188.7k req/s	46.59%	16.87%
3200	186.2k req/s	44.4%	16.87%	187.5k req/s	46.57%	16.35%

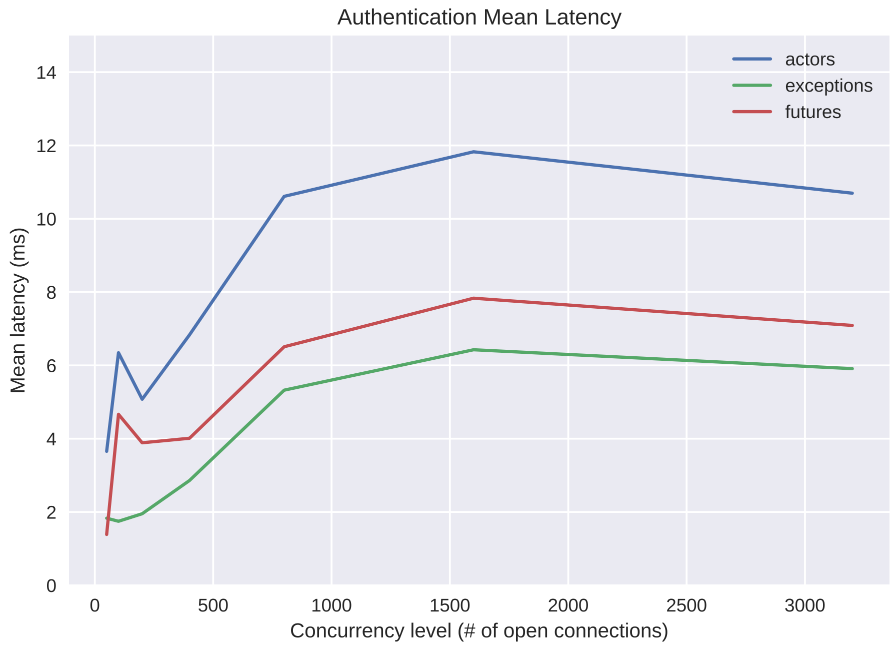

Figure 4.7: Authentication Mean Latency for a Realistic Request Split

Mean Latency In the realistic request split exceptions have the lowest mean latency, followed by futures and actors (Figure 4.7 and Figure 4.8). The three reliability mechanisms follow the same performance patterns. Actors and futures have spikes of high mean latency at 100 concurrent connections for authentication, but actors are consistent for authorisation. At 800 concurrently open connections, when the throughput is the highest, exceptions have the lowest mean latency with 5.32ms for authentication and 5.21ms for authorisation, followed by futures with 6.51ms for authentication and 6.61ms for authorisation, and actors with 10.61ms for authentication and 10.34ms for authorisation.

Key Findings

- **The three reliability mechanisms follow similar performance patterns** (Figure 4.5 and 4.7)
- **Exceptions have the highest throughput for authentication and authorisation** with 186,891 and 189,737 requests per second respectively, followed by futures with 156,379 and 156,274 requests per second respectively, and actors with 108,154 and 104,998 requests per second respectively (Figure 4.5 and 4.6)
- Across both security mechanisms and concurrency levels, actors have an average performance decrease of 45.36% and futures of 17% (Table 4.2)
- **Exceptions have the lowest mean latency** at 800 concurrent connections – 5.32ms for authentication and 5.21 for authorisation, compared to futures with 6.51ms for authentication and 6.61ms for authorisation, and actors with 10.61ms for authentication and 10.34ms for authorisation (Figure 4.7 and 4.8)

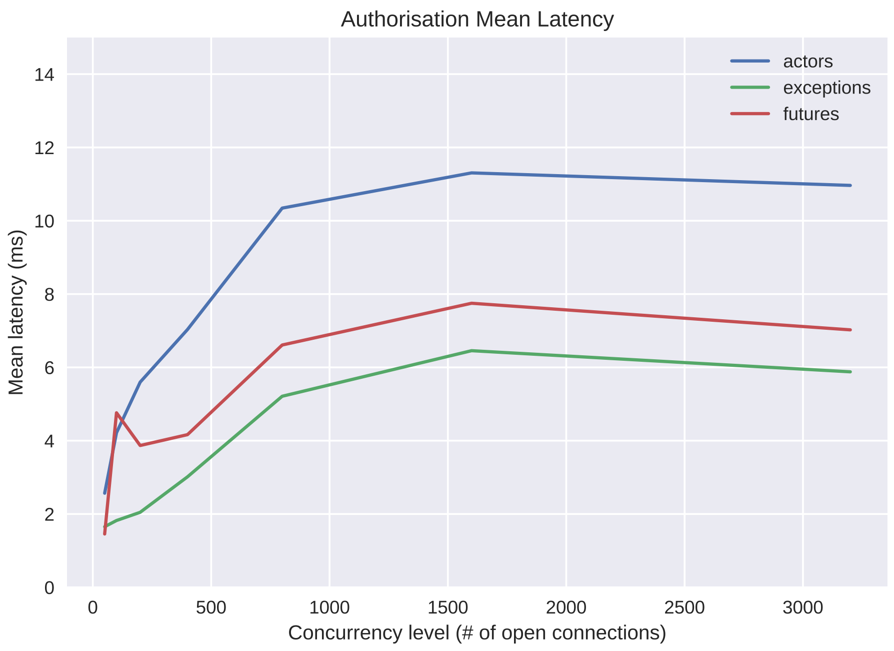

Figure 4.8: *Authorisation Mean Latency for a Realistic Request Split*

4.2.3 Authentication and Authorisation

Although some information has already been presented in the previous experiments on the differences in performance for authentication and authorisation, a direct comparison between the two is provided. Figure 4.9 and Figure 4.10 present graphs on throughput and mean latency for authentication and authorisation for the future model (graphs for the other models are available in Appendix B). It is observed that not only do the two security mechanisms follow the same performance patterns but their lines overlap in multiple points. These results are in line with the expectations outlined in 3.1. Because the two security mechanisms follow the same system architecture and use the exact same components, apart from the authentication and authorisation functions, any potential performance difference would have most likely meant a bottleneck in the security implementation itself, rather than in one of the reliability models. Thus, **actors, exceptions and futures have consistent performance across authentication and authorisation.**

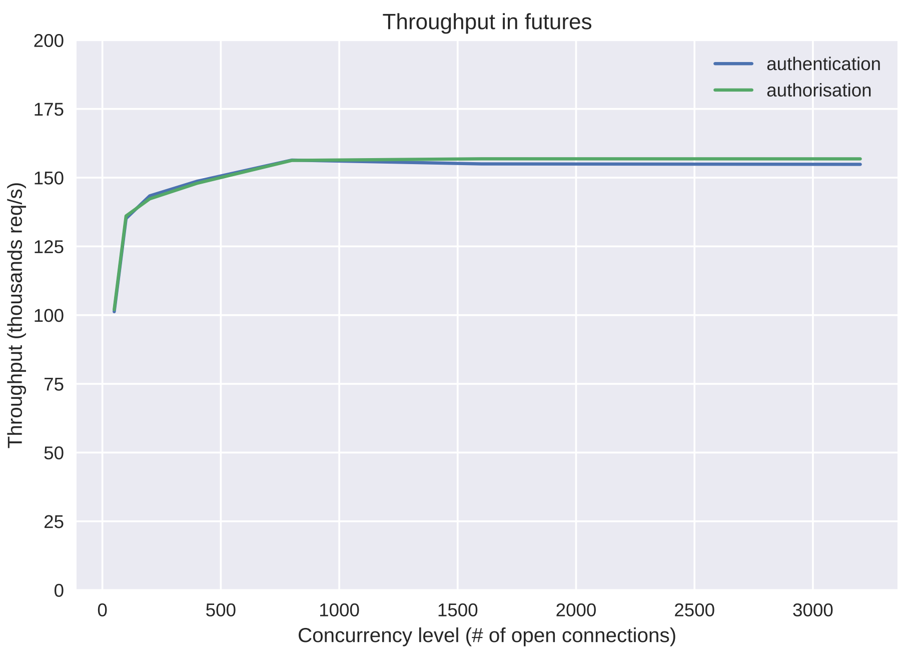

Figure 4.9: *Futures Throughput for Mixed Requests*

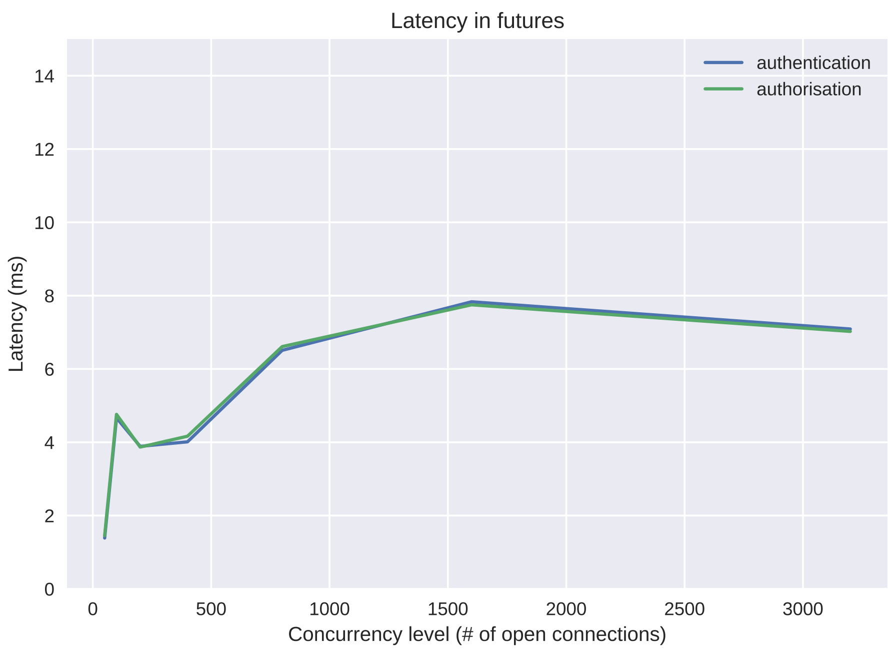

Figure 4.10: *Futures Mean Latency for Mixed Requests*

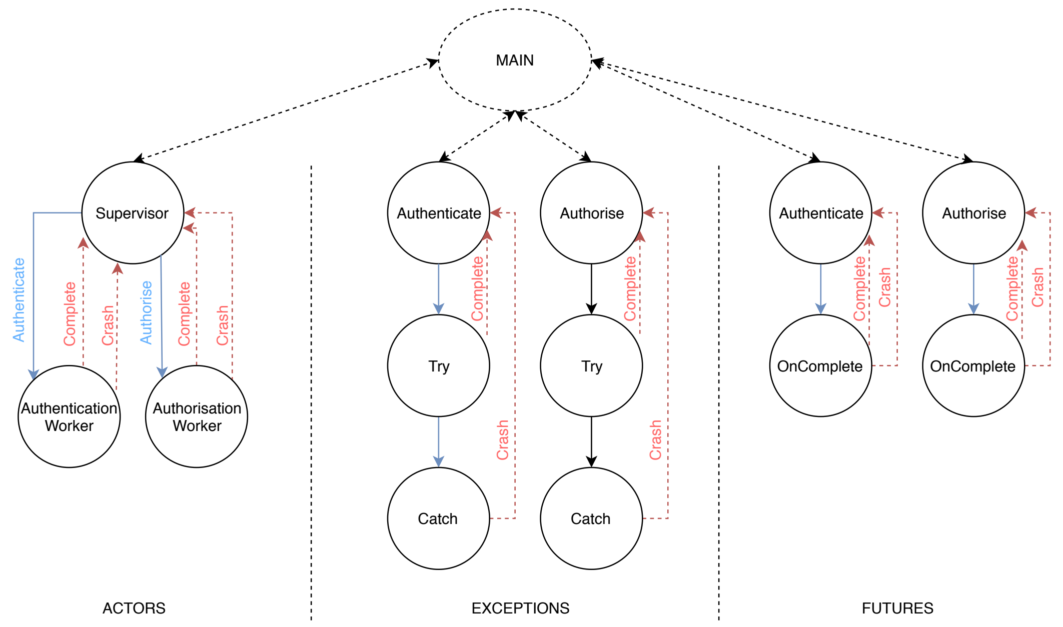

Figure 4.11: State Model Diagram of Actors, Exceptions and Futures

4.3 Programmability

Actors, exceptions and futures are analysed through state modelling and the resulting diagram is presented in Figure 4.11. The MAIN state represents the general flow of the program and the three types of Controllers outlined in Section 3.2.

For the actor model, the program talks to the Supervisor, which spawns either the Authentication or the Authorisation Worker. If the action completes successfully, the child actor returns the appropriate response. If it crashes, the supervisor handles the failure and returns the flow back to MAIN. The messages to restart the child and make it exit with a failure are omitted for the sake of clarity. The resulting number of states for authentication and authorisation with actors is 3.

Exceptions have a starting state which is when the security mechanism action is triggered. For each of the actions, there are two states - try and catch. The try-state is always entered, this is where the authentication or authorisation happens. Upon success, the flow is returned back to the MAIN. If it fails, the catch-state is entered which handles the error and returns flow back to MAIN. This can be further extended with multiple catch clauses, depending on the number of possible exceptions that can be thrown. Moreover, a finally-clause, and consecutively a state, can be added. Multiple catches and finally are omitted for the sake of simplicity. The resulting number of states for authentication and authorisation with exceptions is 6.

Futures work by returning a Future object from a function. An `OnComplete` callback needs to be attached to the authentication or authorisation function. The flow always accesses the security mechanism action and then the respective callback. In the completion function, both the successful and the unsuccessful cases are handled. Upon exiting `OnComplete`, the flow is returned back to MAIN. The resulting number of states for futures is 4.

Key Findings

- **Actors introduce the fewest new states in the program** with 3, followed by futures with 4 and exceptions with 6 (Figure 4.11)
- **Actors have the highest level of programmability and security**

5 Conclusion

This chapter presents a summary of the research problem and the objectives achieved (Section 5.1). A discussion is presented on the limitations of the project and the potential future work (Section 5.2).

5.1 Summary

Authentication and authorisation are an integral part of modern web services. The aim of this research was to compare reliability technologies that handle errors at the application level in these security mechanisms. Actors, exceptions and futures were the selected models, chosen as representatives of vastly different philosophies. Performance in the form of high throughput and low latency is important for end-users of the application. Developers are interested in the programmability of the different models, as it impacts the quality of their work. These two measurements were at the core of this project.

Different application level reliability mechanisms were surveyed and a discussion was presented on exceptions and futures (Section 2.1). The let-it-crash philosophy and the actor model were analysed in Section 2.2. Cybersecurity and its relationship to reliability mechanisms was discussed in Section 2.3.

Based on the literature survey and the required metrics for comparison, an experiment was designed to benchmark the performance and programmability of the three reliability mechanisms (Section 3.1). The performance is measured in terms of throughput and mean latency in different circumstances: different splits of successful and unsuccessful requests, varying numbers of concurrently open connections, and authentication and authorisation. Programmability, in terms of programming complexity and resulting attack surface, is measured by modelling the states of actors, exceptions and futures and counting the number of new states introduced in the program. Based on this experiment a secure web service benchmark with all three reliability mechanisms is designed and implemented in Section 3.2. Using the same architecture for actors, exceptions and futures allowed for performance benchmarks to focus on differences in throughput and latency, originating in the models themselves.

Finally, an evaluation was conducted on the secure web service benchmark, using the load testing tool wrk. The experiments were run on two Glasgow Parallelism Group cluster nodes, running Ubuntu 14.04 with 2 * Intel Xeon E5-2640 2GHz CPUs and 64Gb RAM. One node was running the server and another the load testing tool. The web service benchmark was implemented in Scala 2.12.7 with Akka 2.5.12, Akka-Http 10.1.5, sbt 1.2.6 and JDK 1.8.0. The Java Virtual Machine was restarted between the experiments and warmed-up with 3 runs, whose results were later discarded. All obtained results are the median values of 3 runs.

The first experiment compares the performance of the reliability mechanisms when dealing with only successful and only unsuccessful requests (Section 4.2.1). The results suggest that authentication and authorisation follow similar patterns. Exceptions have the best throughput for successful requests across all concurrency levels. They achieve 196,346 requests per second at 800 connections for authentication, compared to 158,692 requests per second for futures, and 117,477 requests for actors (Figure 4.1). For unsuccessful requests, actors have the best throughput at 50 connections for authentication - 2,360 requests per second, compared to 183 requests per second for exceptions and 333 requests per second for futures. However, futures perform better at 3200 connections with 671 requests per second, compared to 123 requests per second for actors and 365 requests per second for exceptions (Figure 4.2). Throughput degradation is the lowest for actors at 50 connections - 96.41% for authentication, compared to 99.87% for exceptions and 99.67% for futures. However, it is the highest at 3200 connections with 99.9%, compared to futures with 99.57% and exceptions with 99.81% (Table 4.1). Exceptions have the lowest mean latency at 50 connections for both request types - 0.54ms for successful requests and 0.25ms for unsuccessful requests. In contrast, futures have 1.19ms for successful requests and 0.38ms for unsuccessful requests, and actors - 1.67ms for successful requests and 0.63ms for unsuccessful requests. Despite unsuccessful requests having low mean latency, they dramatically decrease the throughput in all three models (Figure 4.1 and 4.2).

In the second experiment successful and unsuccessful requests are mixed in an 89/11% split respectively (Section 4.2.2). The three reliability mechanisms have similar performance patterns. Exceptions have the highest authentication throughput with 186,891 requests per second and the lowest mean latency with 5.32ms at 800 concurrent connections, followed by futures with 156,379 requests per second and 6.51ms,and actors with 108,154 and 10.61ms (Figure 4.5 and 4.7). Actors have an average throughput decrease of 45.36%, and futures - 17%.

The final performance experiment looks at performance differences in the three reliability models for authentication and authorisation (Section 4.2.3). Figure 4.9 and 4.10 show the same patterns for throughput and latency across the different security mechanisms. This is expected as authentication and authorisation have the same architecture and system structure. A performance discrepancy would most likely mean a bottleneck in one of the security functions, rather than the reliability model.

Through state modelling we analyse the number of states that the three reliability mechanisms introduce in the program (Section 3.1.2). Actors have the fewest number of states with 3, followed by futures with 4, and exceptions with 6 (Figure 4.11). States are important because they are units of the program which developers can reason about. Ensuring the security and correctness of each state ensures the robustness of the program. Fewer states makes it easier to write correct software and reduces the possible attacking vectors for potential attackers. Thus, actors have the highest level of programmability and security.

Exceptions have the best performance, in terms of throughput and latency, but introduce the most new states in the program. Actors are at the other end of the spectrum with the worst performance but the fewest new states. Futures are in the middle in both categories. There is a clear inverse relationship and a trade-off between performance and programmability. When throughput and latency are not that important, actors are the recommended choice as authentication and authorisation can be securely implemented more quickly. If performance is a priority, though, futures and exceptions can provide a better alternative for the cost of more development time.

5.2 Future Work

Despite the obtained results and the achievements covered in this dissertation, the project has some limitations that can be addressed in a future study.

Firstly, the language and framework of choice for the secure web application benchmark can be performance bottlenecks. In the currently proposed setting, the three reliability mechanisms follow the same patterns for throughput and mean latency. However, all models struggle to have performance higher than 200 thousand requests per second for the mixed request scenario. This can be due to the nature of actors, exceptions and futures but can also mean that Akka-Http is unable to handle loads larger than the results highlighted in Section 4.2.2. Thus, different web server frameworks might have considerably different performance. Alternatively, the three reliability mechanisms can be implemented in different, more idiomatic languages. For example, the actor model can be designed as an Erlang web service. Processes in BEAM (the Erlang Virtual Machine) are much more lightweight than the Akka actor implementation, which runs in the Java Virtual Machine. Similarly, futures might have better performance in a purely functional language, such as Haskell.

Another possible limitation is the fixed number of supervisors in the actors' implementation of the secure web app benchmark. The arbitrary number 10 might not provide the best fan-out between the authentication and authorisation workers and as such might be a performance bottleneck. An additional experiment can be carried out with a varying number of starting supervisors or with a dynamically growing and shrinking list. This parameter tuning would find the most optimal number of supervising actors for high throughput and low latency. A caveat in this experiment would be the different levels of concurrently open connections that can be explored. The most optimal number of supervisors might depend on the number of connections.

Finally, the choice of load-testing tool matters, as showcased in Appendix A. The results for throughput and latency for each experiment are obtained with wrk and implementation quirks or bugs might have influenced them. There is no reason to suspect this has happened, but extensive research can be made into different load-testing tools, their performance benchmarking the same web service, advantages and disadvantages. This would show when each potential choice would make the most sense and when each of them might be a good fit for a study, such as the one presented in this dissertation.

A Comparing Load-testing Tools

A.1 Context

During the course of the project two different benchmarking tools were used for conducting the performance experiments - Apache Bench (Apache Bench 2019) and wrk (wrk 2019). In the early stages of the project a short research was carried out comparing different load testing frameworks, such as Apache JMeter (Apache JMeter 2019) and http_load (http-load 2019), as well as ab. All of these frameworks share the same basic functionality - sending a number of requests to a web server with a fixed number of concurrently open connections and then measuring different statistics, related to the responses they receive. All tools measure standard metrics, such as throughput and latency. Some focus more on the spread of the latency data, offering measurements of the different quantiles and percentages. Others pay more attention to failed requests, received HTTP status codes, headers, etc. Ultimately, Apache Bench was the selected tool of choice, because of being:

- lightweight
- widely available in Linux distributions, as part of the apache2-utils package
- written in C, which guaranteed no performance bottlenecks when running against a service running in the JVM
- able to offer all statistical data necessary for the experiment

However, several disadvantages started to show up during the experiment evaluation. Firstly, Apache Bench does not provide a mechanism for sending different requests as part of the same benchmark. This would make it very difficult to perform the test with mixed successful and unsuccessful requests and would require usage of GNU's Parallel tool (Tange 2011), as a workaround. Furthermore, Apache Bench does not offer a way to format the data in a structured manner, e.g. CSV or JSON. This would raise the need for writing a parsing script to gather the data in a format easy for analysis. However, the complexity associated with this task would increase when running the benchmark through Parallel, as that would create multiple output files.

A weird result trend was discovered when running Apache Bench experiments with a fixed number of requests. There was an evident quantum effect in the throughput for different scenarios. The requests per second would ultimately always vary between numbers circa 6000 and circa 3000. There would be very few values in between these two measures. Furthermore, as the experiment required 3 runs of each particular scenario, sometimes identical scenarios would have values around 6000 and other times around 3000. Figure A.1 shows the problem happening when running the experiments for actors with successful and unsuccessful requests. This was worrying, as it suggested incorrect results. Apache Bench was identified as the main culprit and a change in the benchmarking tool was necessary.

At this point, a second round of research was conducted. This exposed what might be the underlying issues behind some of the weird results, achieved with Apache Bench. Ab is quite an old tool. It does not support new HTTP versions, it is single-threaded and has been designed to test the capabilities of Apache servers, while nowadays there are much more scalable technologies, such as nginx (NGINX 2019). During this research several more tools were explored, including siege (Siege 2019), perfmeter (Java Perfmeter 2019) and wrk. Wrk was selected because of its

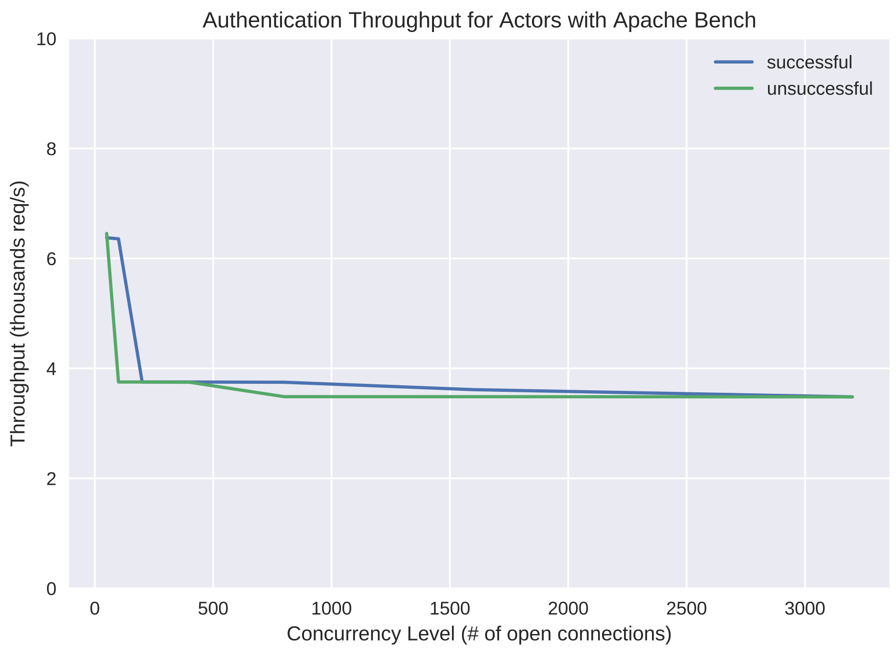

Figure A.1: *Quantum Effect in Actors Throughput with Apache Bench*

scalable design and Lua JIT scripting, which allows benchmark modifications during runtime without performance hits. This solved the issues introduced by Apache Bench, namely:

- multithreaded by design
- utilises modern event notification systems like epoll and kqueue to improve performance
- varying the requests allowed us to achieve the desired 89/11% request split
- outputting the data in a CSV format

After running several small scale tests to ensure the quantum effect does not happen, wrk was the chosen tool for the remainder of the project and the one with which the main evaluation results were achieved.

A.2 Comparing ab and wrk

A short comparative experiment was conducted on the performance of Apache Bench and wrk. The two load-testing tools ran in a scenario with 100% successful requests for 5 minutes with the exceptions reliability mechanism. For the sake of equality, wrk was run in single-threaded mode. The results are presented in figure A.2.

The difference in throughput is clearly in favour of wrk. The requests per second achieved by Apache Bench are almost 20 times less across all concurrency levels. Furthermore, the one extreme of the quantum effect, outlined in Section A.1 is again observed - all values are circa 6000 requests per second. The difference in performance would be even larger if wrk's multi-threaded capabilities were enabled.

The conclusion from this short experiment is that the throughput that Apache Bench reports is extremely unreliable and does not objectively represent the limits of tested web service. We do

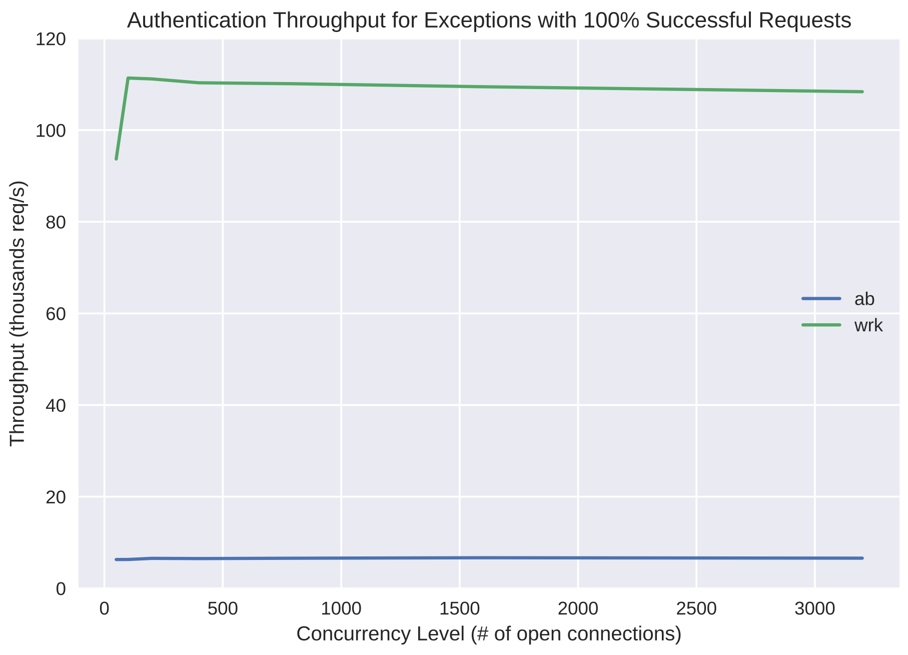

Figure A.2: *Comparing ab and wrk throughput*

not have any guarantees that wrk reports accurate values but it is definitely a better performer than ab.

B | Authentication and Authorisation Performance Experiments Results

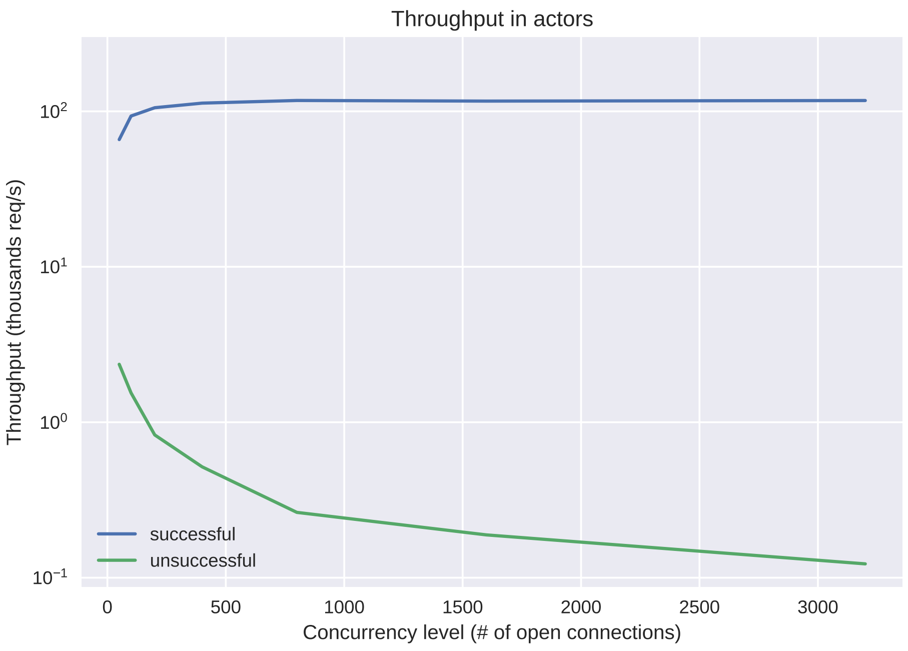

Figure B.1: *Throughput for Actors Authentication with Different Request Types*

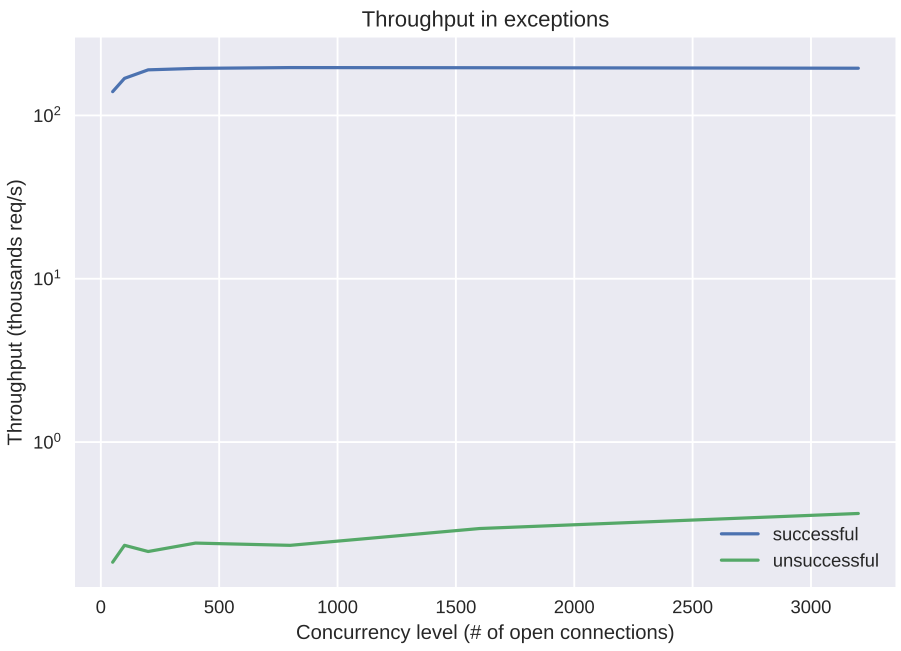

Figure B.2: *Throughput for Exceptions Authentication with Different Request Types*

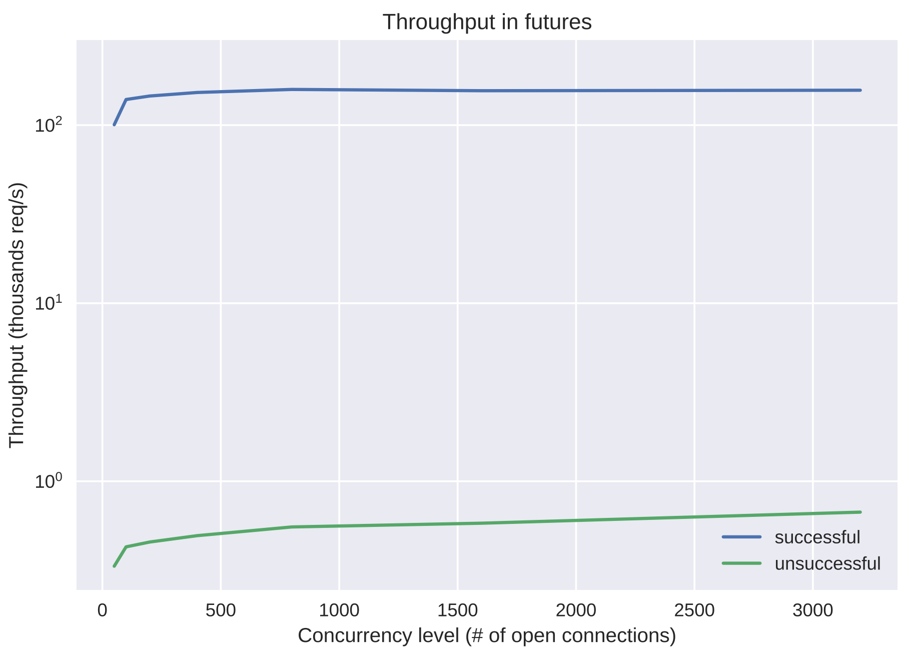

Figure B.3: *Throughput for Futures Authentication with Different Request Types*

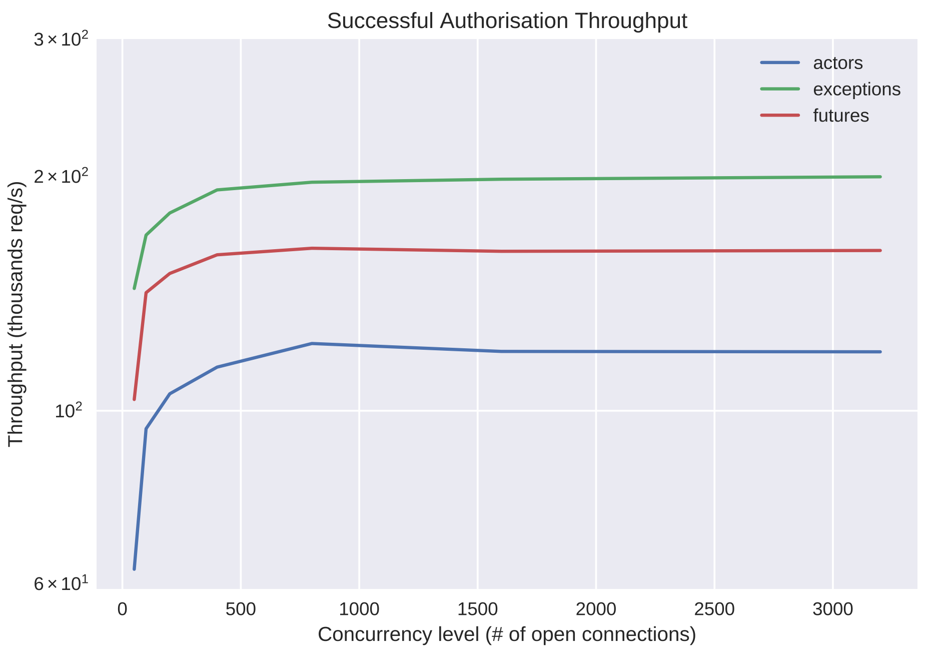

Figure B.4: *Throughput for Authorisation with 100% Successful Requests*

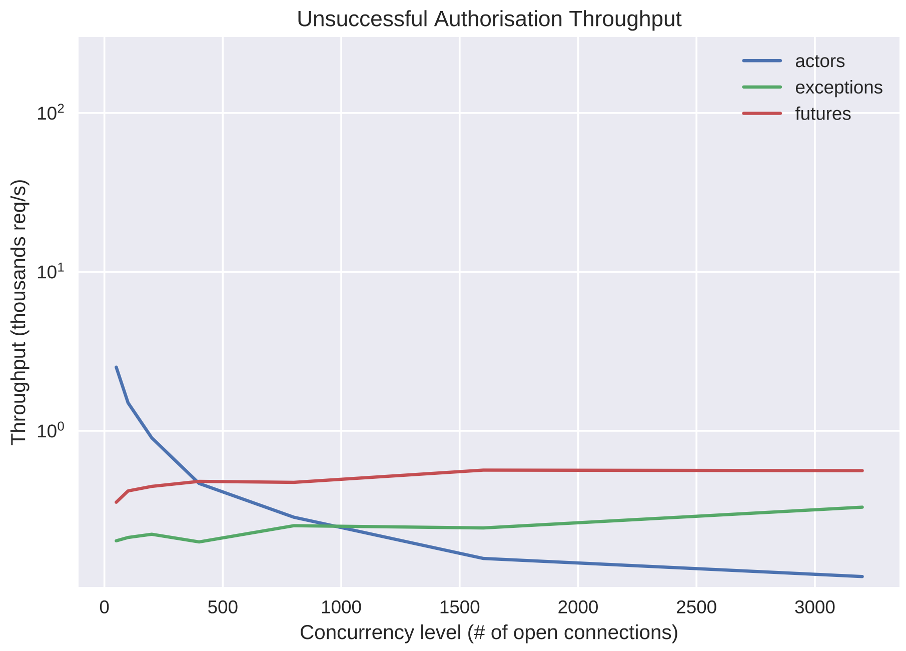

Figure B.5: *Throughput for Authorisation with 100% Unsuccessful Requests*

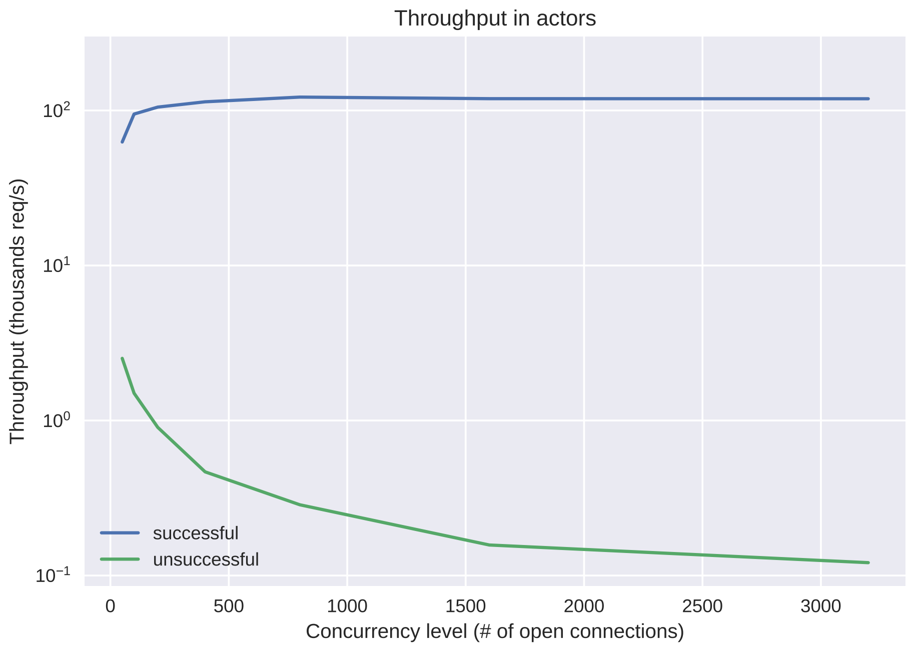

Figure B.6: *Throughput for Actors Authorisation with Different Request Types*

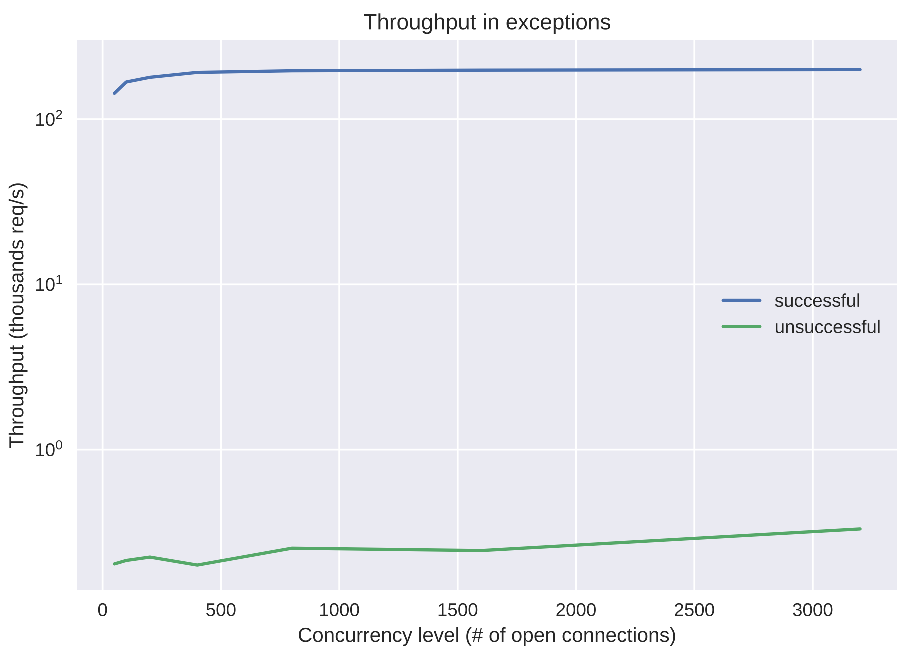

Figure B.7: *Throughput for Exceptions Authorisation with Different Request Types*

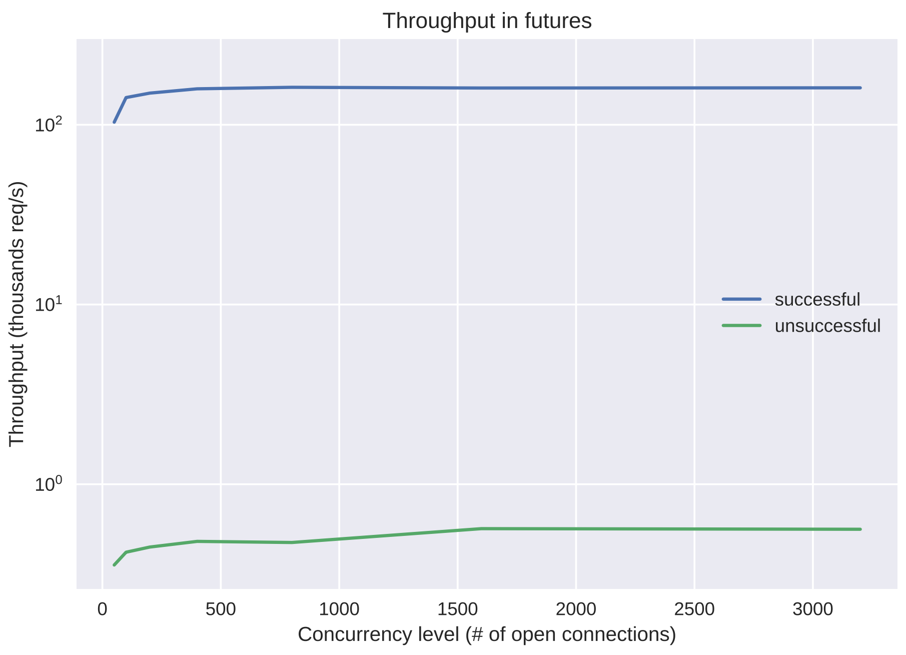

Figure B.8: *Throughput for Futures Authorisation with Different Request Types*

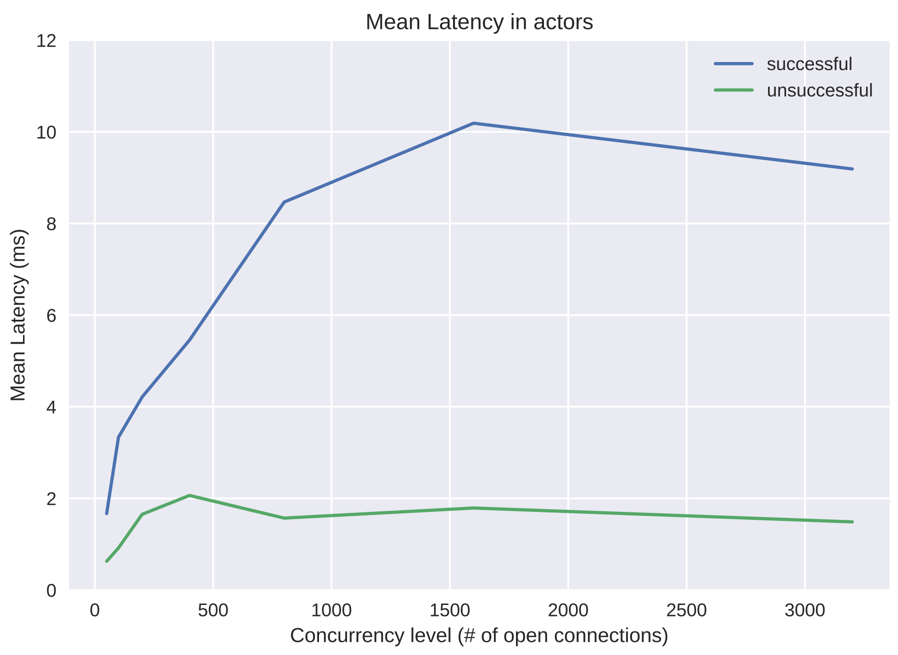

Figure B.9: *Mean Latency for Actors Authentication with Different Request Types*

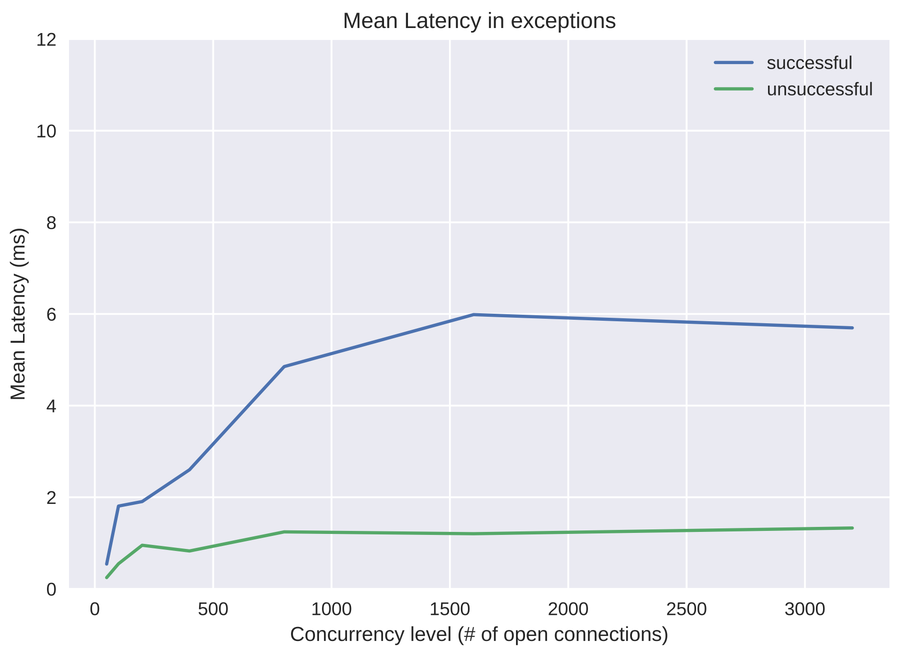

Figure B.10: *Mean Latency for Exceptions Authentication with Different Request Types*

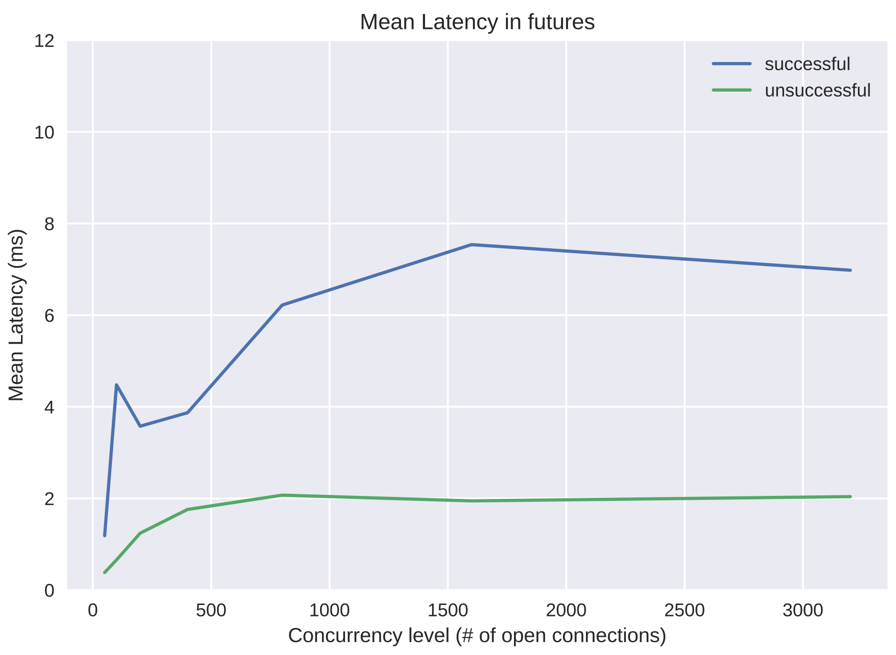

Figure B.11: *Mean Latency for Futures Authentication with Different Request Types*

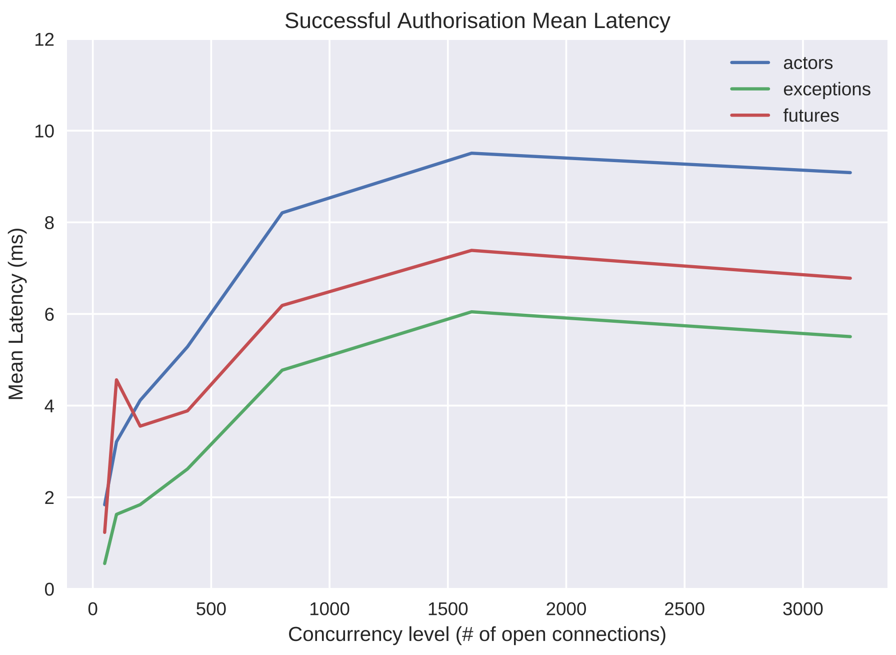

Figure B.12: *Mean Latency for Authorisation with 100% Successful Requests*

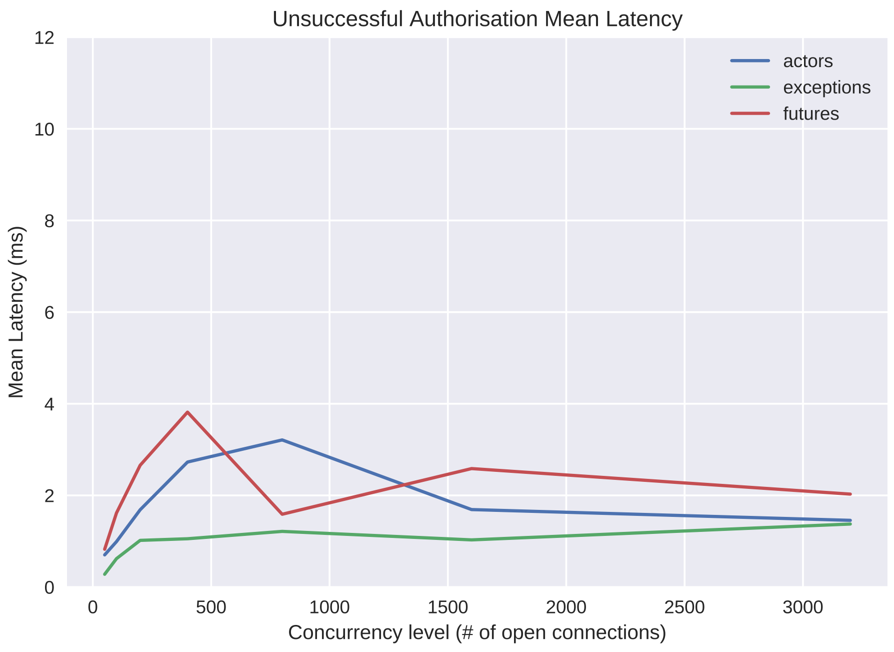

Figure B.13: *Mean Latency for Authorisation with 100% Unsuccessful Requests*

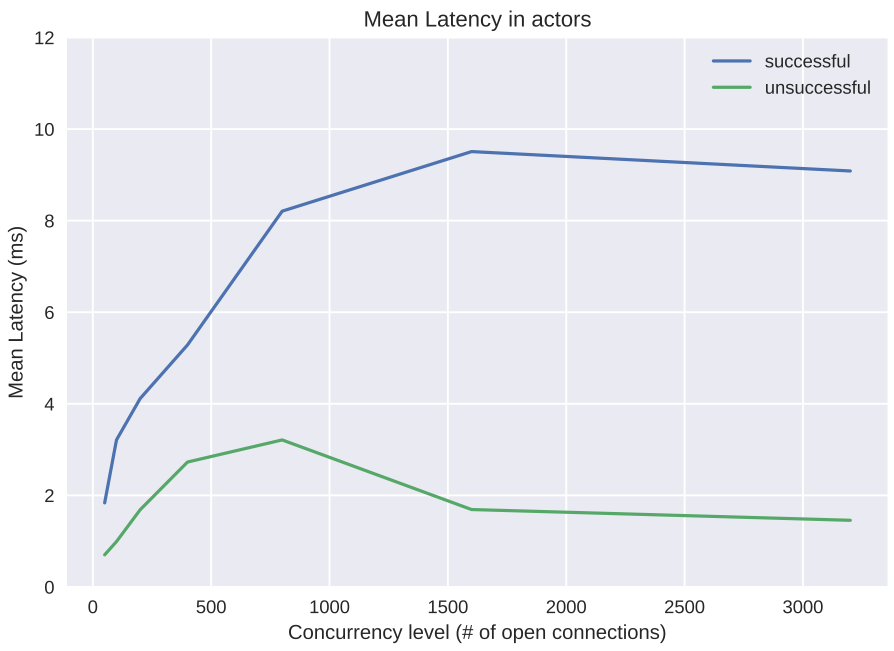

Figure B.14: *Mean Latency for Actors Authorisation with Different Request Types*

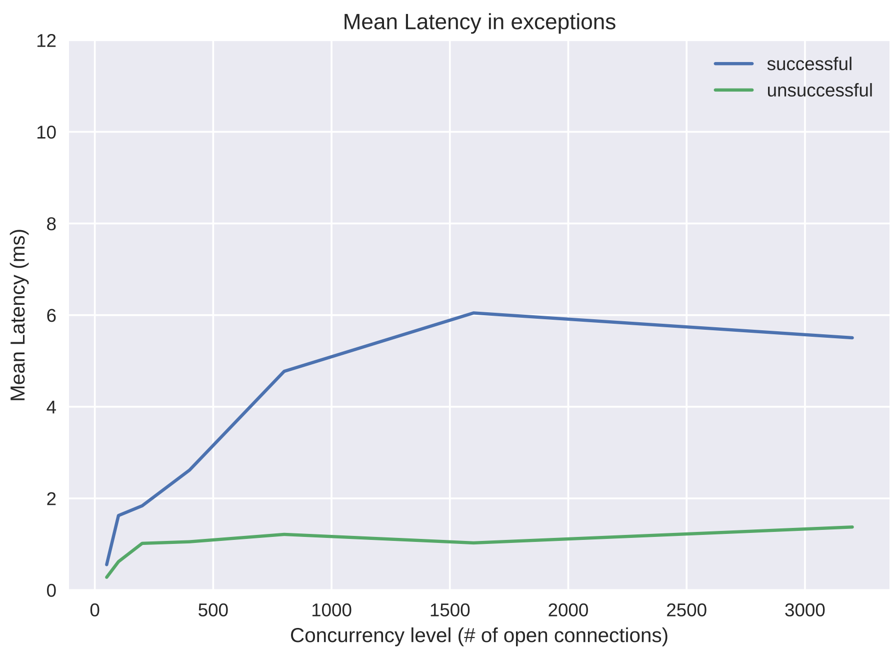

Figure B.15: *Mean Latency for Exceptions Authorisation with Different Request Types*

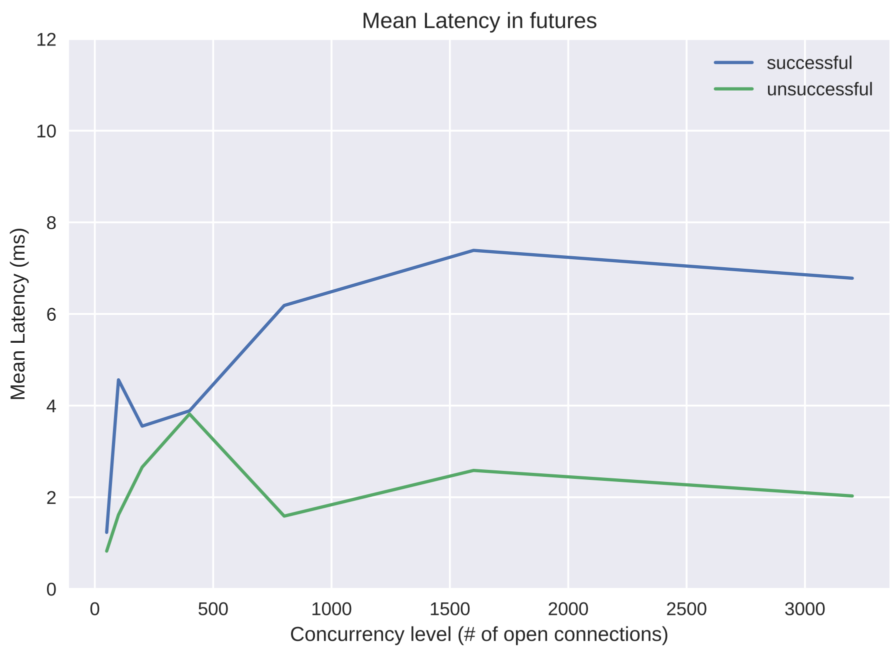

Figure B.16: *Mean Latency for Futures Authorisation with Different Request Types*

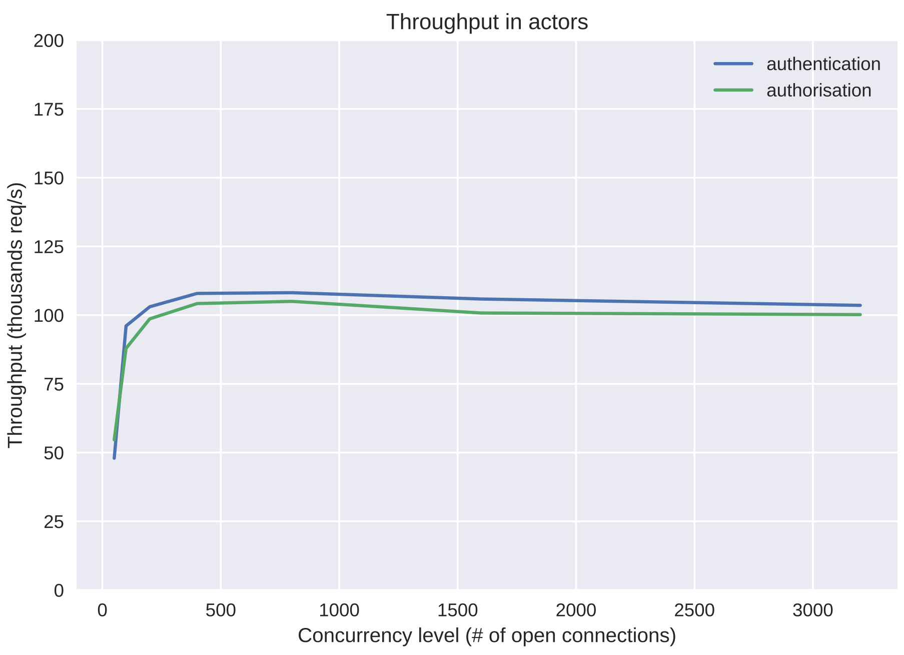

Figure B.17: *Actor Throughput for Mixed Requests*

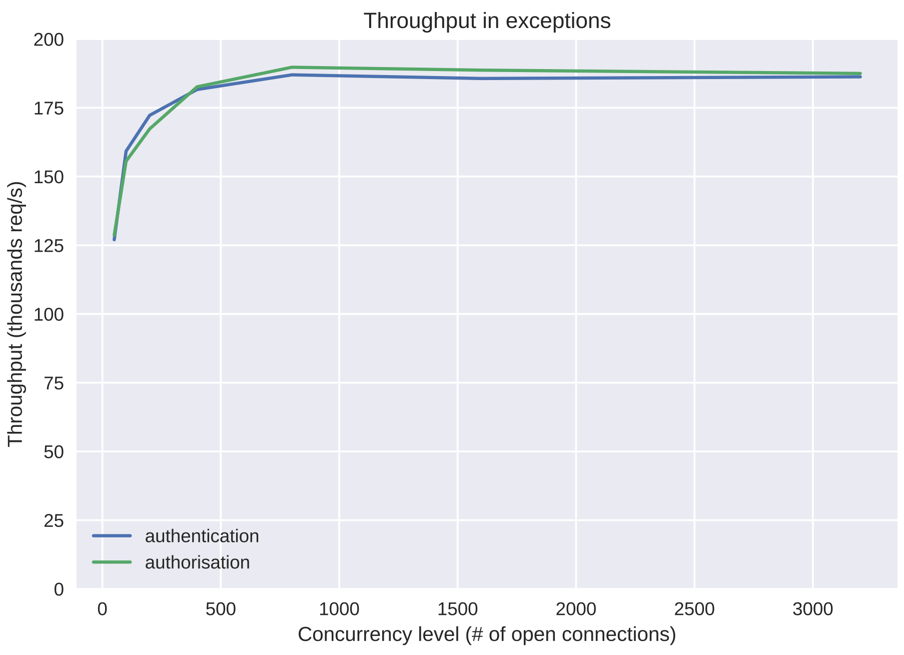

Figure B.18: *Exceptions Throughput for Mixed Requests*

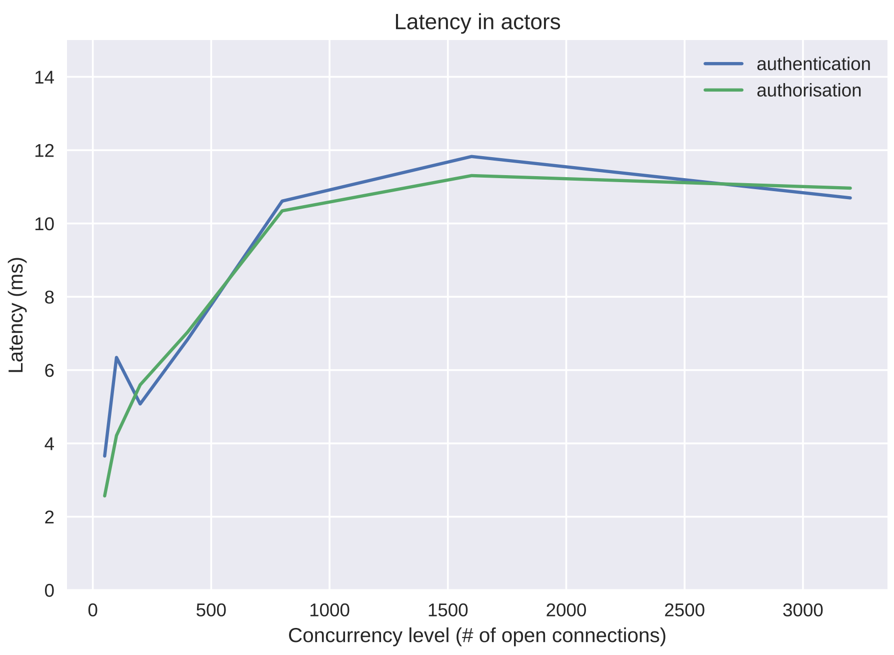

Figure B.19: *Actor Mean Latency for Mixed Requests*

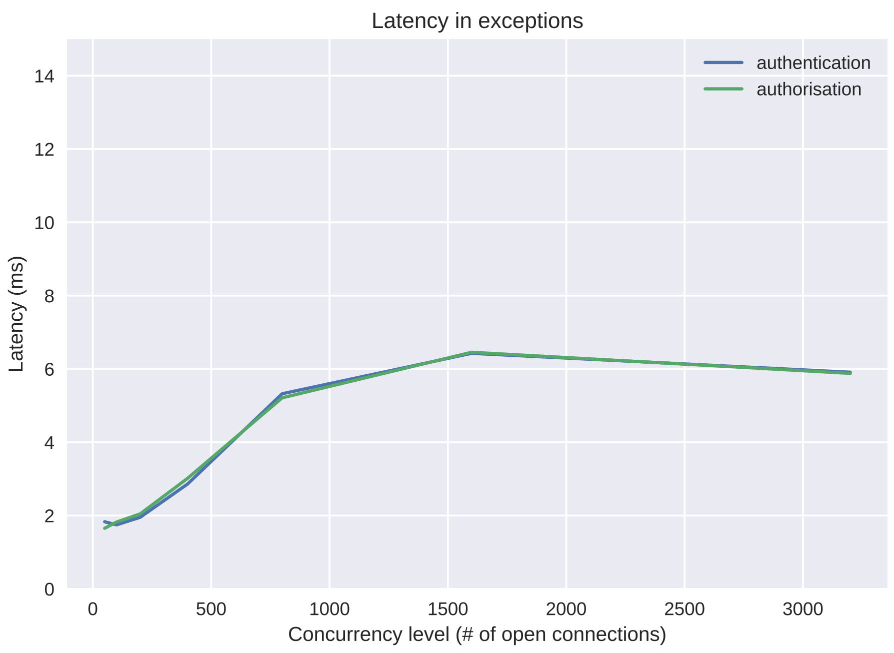

Figure B.20: *Exceptions Mean Latency for Mixed Requests*

5 Bibliography

Akka HTTP 2019. Akka HTTP, 2019. URL `https://doc.akka.io/docs/akka-http/10.0.9/scala/http/`. Accessed: 23-03-2019.

Akka.NET 2019. Akka.NET Documentation | Akka.NET Documentation, 2019. URL `https://getakka.net/`. Accessed: 15-01-2019.

Apache Bench 2019. ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4, 2019. URL `https://httpd.apache.org/docs/2.4/programs/ab.html`. Accessed: 21-03-2019.

Apache JMeter 2019. Apache JMeter - Apache JMeterâĎć, 2019. URL `https://jmeter.apache.org/`. Accessed: 21-03-2019.

J. Armstrong. Making reliable distributed systems in the presence of software errors, 2003.

J. Armstrong. A history of erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1. ACM, 2007.

bagwell 2010. Scala at LinkedIn | The Scala Programming Language, 2010. URL `https://www.scala-lang.org/old/node/6436`. Accessed: 23-03-2019.

Baron Schwartz 2015. The Factors That Impact Availability, Visualized. `https://www.vividcortex.com/blog/the-factors-that-impact-availability-visualized`, 2015. Accessed: 07-10-2018.

B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 1st edition, 2016. ISBN 149192912X, 9781491929124.

Blesson Paul 2016. Announcing Neutrino for Load Balancing and L7 Switching, 2016. URL `https://www.ebayinc.com/stories/blogs/tech/announcing-neutrino-for-load-balancing-and-l7-switching/`. Accessed: 23-03-2019.

N. Briscoe. Understanding the osi 7-layer model. *PC Network Advisor*, 120(2), 2000.

Deepak Vasthimal 2016. Scalable and Nimble Continuous Integration for Hadoop Projects, 2016. URL `https://www.ebayinc.com/stories/blogs/tech/scalable-and-nimble-continuous-integration-for-hadoop-projects/`. Accessed: 23-03-2019.

C. Dony, C. Urtado, and S. Vauttier. Advanced topics in exception handling techniques. In C. Dony, J. L. Knudsen, A. Romanovsky, and A. Tripathi, editors, *Advanced Topics in Exception Handling Techniques*, chapter Exception Handling and Asynchronous Active Objects: Issues and Proposal, pages 81–100. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 3-540-37443-4, 978-3-540-37443-5. URL `http://dl.acm.org/citation.cfm?id=2124243.2124250`.

Eclipse Vert.x 2019. Eclipse Vert.x, 2019. URL `https://vertx.io/`. Accessed: 15-01-2019.

Erlang 2018. Erlang Programming Language. https://www.erlang.org/, 2018. Accessed: 03-10-2018.

C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL http://dl.acm.org/citation.cfm?id=1624775.1624804.

http-load 2019. http_load. https://acme.com/software/http{_}load/, 2019. Accessed: 21-03-2019.

IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990. doi: 10.1109/IEEESTD.1990.101064.

Java Perfmeter 2019. Java Perfmeter - JPerfmeter, 2019. URL http://jperfmeter.sourceforge.net/. Accessed: 21-03-2019.

Joe Armstrong 2013. Joe Armstrong blogpost on Erlang and Elixir, 2013. URL https://joearms.github.io/published/2013-05-31-a-week-with-elixir.html. Accessed: 08-10-2018.

R. Kissel. Nist ir 7298 revision 2: Glossary of key information security terms. *National Institute of Standards and Technology. May. Accessed July*, 7:2013, 2013.

Kubernetes 2019. Production-Grade Container Orchestration - Kubernetes, 2019. URL https://kubernetes.io/.

B. Liskov, L. Shrira, B. Liskov, and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *ACM SIGPLAN Notices*, 23(7):260–267, jul 1988. ISSN 03621340. doi: 10.1145/960116.54016. URL http://portal.acm.org/citation.cfm?doid=960116.54016.

S. Mare, M. Baker, and J. Gummeson. A study of authentication in daily life. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, pages 189–206, Denver, CO, 2016. USENIX Association. ISBN 978-1-931971-31-7. URL https://www.usenix.org/conference/soups2016/technical-sessions/presentation/mare.

Marius Eriksen 2012. Scala at Twitter, 2012. URL http://twitter.github.io/effectivescala/. Accessed: 23-03-2019.

J. A. Miller, C. Bowman, V. G. Harish, and S. Quinn. Open Source Big Data Analytics Frameworks Written in Scala. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 389–393. IEEE, jun 2016. ISBN 978-1-5090-2622-7. doi: 10.1109/BigDataCongress.2016.61. URL http://ieeexplore.ieee.org/document/7584967/.

Mozilla Developer Network 2018. Using promises - JavaScript | MDN, 2018. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises. Accessed: 08-11-2018.

Netlifx 2018. Chaos Monkey. https://netflix.github.io/chaosmonkey/, 2018. Accessed: 03-10-2018.

NGINX 2019. NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy, 2019. URL https://www.nginx.com/. Accessed: 21-03-2019.

Philipp Haller and Stephen Tu 2013. The Scala Actors API | Scala Documentation. https://docs.scala-lang.org/overviews/core/actors.html, 2013. Accessed: 08-10-2018.

Proto.Actor 2019. Welcome to Proto.Actor | ProtoActor, 2019. URL `http://proto.actor/`. Accessed: 15-01-2019.

R. Reed. Scaling to millions of simultaneous connections. *Erlang Factory SF*, 2012.

J. Reschke. The 'Basic' HTTP Authentication Scheme. Technical report, IETF, sep 2015. URL `https://www.rfc-editor.org/info/rfc7617`.

Scala Play 2019. Play Framework - Build Modern & Scalable Web Apps with Java and Scala, 2019. URL `https://www.playframework.com/`. Accessed: 23-03-2019.

Scalatra 2019. Scalatra, 2019. URL `http://scalatra.org/`. Accessed: 23-03-2019.

J. Siedersleben. Advanced topics in exception handling techniques. In C. Dony, J. L. Knudsen, A. Romanovsky, and A. Tripathi, editors, *Advanced Topics in Exception Handling Techniques*, chapter Errors and Exceptions – Rights and Obligations, pages 275–287. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 3-540-37443-4, 978-3-540-37443-5. URL `http://dl.acm.org/citation.cfm?id=2124243.2124263`.

Siege 2019. Siege Home, 2019. URL `https://www.joedog.org/siege-home/`. Accessed: 21-03-2019.

O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1): 42–47, Feb 2011. doi: 10.5281/zenodo.16303. URL `http://www.gnu.org/s/parallel`.

The Go Programming Language 2018. Frequently Asked Questions (FAQ) - The Go Programming Language. `https://golang.org/doc/faq#exceptions`, 2018. Accessed: 07-10-2018.

E. Union. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union*, L119:1–88, May 2016. URL `http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC`.

W3Techs 2019. Usage Statistics and Market Share of Server-side Programming Languages for Websites, March 2019. `https://w3techs.com/technologies/overview/programming_language/all`, 2019. Accessed: 23-03-2019.

W. Weimer and G. C. Necula. Exceptional situations and program reliability. *ACM Trans. Program. Lang. Syst.*, 30(2):8:1–8:51, Mar. 2008. ISSN 0164-0925. doi: 10.1145/1330017.1330019. URL `http://doi.acm.org/10.1145/1330017.1330019`.

wrk 2019. wg/wrk - Modern HTTP benchmarking tool, 2019. URL `https://github.com/wg/wrk`. Accessed 21-03-2019.