Master Thesis
Software Engineering
Thesis no: MCS-2014-NN
09 2014

Migration from blocking to non-blocking web frameworks

Guidelines for profits assessment

Mateusz Bilski

Dept. Computer Science & Engineering
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Department of Computer Science & Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 15 weeks of full-time studies.

Contact Information:
Author: Mateusz Bilski
E-mail: mateusz.bilski@gmail.com

University advisor:
Dr. Ludwik Kuźniarz
Dept. Computer Science & Engineering

Dept. Computer Science & Engineering
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se/didd
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context. The problem of performance and scalability of web applications is challenged by most of the software companies. It is difficult to maintain the performance of a web application while the number of users is continuously increasing. The common solution for this problem is scalability. A web application can handle incoming and outgoing requests using blocking or non-blocking Input/Output operation. The way that a single server handles requests affects its ability to scale and depends on a web framework that was used to build the web application. It is especially important for Resource Oriented Architecture (ROA) based applications which consist of distributed Representational State Transfer (REST) web services. This research was inspired by a real problem stated by a software company that was considering the migration to the non-blocking web framework but did not know the possible profits.
Objectives. The objective of the research was to evaluate the influence of web framework's type on the performance of ROA based applications and to provide guidelines for assessing profits of migration from blocking to non-blocking JVM web frameworks.
Methods. First, internet ranking was used to obtain the list of the most popular web frameworks. Then, the web frameworks were used to conduct two experiments that investigated the influence of web framework's type on the performance of ROA based applications. Next, the consultations with software architects were arranged in order to find a method for approximating the performance of overall application. Finally, the guidelines were prepared based on the consultations and the results of the experiments.
Results. Three blocking and non-blocking highly ranked and JVM based web frameworks were selected. The first experiment showed that the non-blocking web frameworks can provide performance up to 2.5 times higher than blocking web frameworks in ROA based applications. The experiment performed on existing application showed average 27% performance improvement after the migration. The elaborated guidelines successfully convinced the company that provided the application for testing to conduct the migration on the production environment.
Conclusions. The experiment results proved that the migration from blocking to non-blocking web frameworks increases the performance of web application. The prepared guidelines can help software architects to decide if it is worth to migrate. However the guidelines are context depended and further investigation is needed to make it more general.

Keywords: performance, migration, non-blocking, blocking, guidelines, Resource Oriented Architecture, ROA, JVM, Java

Acknowledgments

I would like to thank my supervisor dr. Ludwik Kuźniarz for constructive comments, valuable advises, continuous guidance and patience.

I am also giving my great thanks to software engineers and architects, from the company which provided application for testing, who were supporting me during the experiments and had a significant contribution in the preparation of the guidelines.

Glossary[1]

API Application Programming Interface defines set of rules in which programs are communicating with each other. For REST services API is a specification of remote endpoints that are exposed to the clients.

AWS Amazon Web Services is a collection of remote computing services that together make up a cloud computing platform, offered over the Internet by Amazon.com.

B, NB Blocking, Non-blocking are two approaches for handling incoming and outgoing requests by the web application server.

Chef Configuration management tool wrote in Ruby and Erlang.

CPU Central Processing Unit is the hardware within a computer that carries out the instructions of a computer program by performing the basic arithmetical, logical, and input/output operations of the system.

Epoll, Kqueue Epoll is a Linux kernel system call, a scalable I/O event notification mechanism, meant to replace the older POSIX Select and Poll system calls. Kqueue is the equivalent system call for Epoll in FreeBSD.

HTTP Hypertext Transfer Protocol is an web application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web.

I/O Input/Output is the communication between an information processing system and the outside world, possibly a human or another information processing system.

[1]The Wikipedia (http://en.wikipedia.org/wiki/Main_Page) is a reference for most of the definitions.

JDK Java Development Kit is an implementation of Java released by Oracle Corporation in the form of a binary product aimed at Java developers. The JDK has as its primary components a collection of programming tools.

JSON JavaScript Object Notation is an open standard format that uses human-readable text to transmit data objects consisting of key-value pairs. It is used primarily to transmit data between a server and web application, as an alternative to XML.

JVM Java Virtual Machine is a process virtual machine that can execute Java bytecode. It is the code execution component of the Java platform.

LDAP Lightweight Directory Access Protocol is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services over a network.

Netstat is a command-line tool that displays network connections (both incoming and outgoing), routing tables, and a number of network interfaces and network protocol statistics.

NIO Non-blocking I/O is a collection of Java programming language APIs that offer features for intensive I/O operations.

REST Representational State Transfer is an architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements.

RFC Requests for Comments are publications of the Internet Engineering Task Force (IETF).

RMI Remote Method Invocation is a Java API that performs the object-oriented equivalent of remote procedure calls, with support for direct transfer of serialized Java classes and distributed garbage collection.

ROA Resource Oriented Architecture is a style of software architecture and programming paradigm for designing and developing software in the form of resources with REST interfaces. These resources are software components (discrete pieces of code and/or data structures) which can be reused for different purposes

RPS Requests Per Second is the number of requests that an web application is able to handle within one second without failure.

RT Response Time is time between sending a request and receiving a response.

Ruby Dynamic, reflective, object-oriented, general-purpose programming language.

SOAP Simple Object Access Protocol is a protocol specification for exchanging structured information in the implementation of web services in computer networks. It relies on XML for its message format, and usually relies on other application layer protocols, most notably Hypertext Transfer Protocol, for message negotiation and transmission.

URI Uniform Resource Identifier is a string of characters used to identify a name of a resource. Such identification enables interaction with representations of the resource over a network, typically the World Wide Web, using specific protocols. The most common form of URI is URL.

URL Uniform Resource Locator is a specific character string that constitutes a reference to a resource. Most web browsers display the URL of a web page above the page in an address bar.

XML Extensible Markup Language is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It is defined in the XML 1.0 Specification produced by the W3C, and several other related specifications, all free open standards.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The most popular web applications are visited by high number of users simultaneously and because of that they should provide high performance. An example of such web application is Twitter, which according to the statistics[1] in 2013 had 255 million active users, and 500 million messages were sent per day. In 2011 Twitter had *only* 100 million active users. Another web application, Facebook, in 2013 had over 1 billion active users and every 20 minutes 3 million messages were sent[2]. Facebook noted 22% increase in users from 2012 to 2013. In 2013 Instagram reached 150 million users. Instagram gained 50 million more users within six months[3].

A web application can be qualified as a high performing web application, when the time required to process a request is low and acceptable by the users even during a heavy load. A heavy load occurs when lots of users are using the web application at the same time. It is difficult to keep the time needed to process the requests on the same low level while the number of users is increasing.

The key to maintain the performance of web application is scalability. It is defined as ability to increase the number of requests that can be handled simultaneously, when new resources are added [1]. There are two main approaches to scale a web application:

- to scale horizontally - to add new servers
- to scale vertically - to add resources to a single server (typically CPU or memory)

The horizontal scaling is more expensive and may be limited due to synchronization overhead [2]. The vertical scaling can be applied only when web application is designed in such a way that it is able to fully utilize resources [3]. The way that a server handles requests affects its ability to scale vertically. There are two ways to handle a request: using blocking or non-blocking Input/Output

[1] https://about.twitter.com/company
[2] http://www.statisticbrain.com/facebook- statistics/
[3] http://blog.instagram.com/post/60694542173/150-million

(I/O) operations. Web frameworks[4] that uses non-blocking approach ensures that resources are fully utilized [2] and provide higher performance than blocking web frameworks.

The web application[5] with non-blocking web frameworks might be able to handle more requests using fewer servers than with blocking web frameworks [4]. For companies that are looking to increase the performance of their web application which uses blocking web frameworks it might be more cost-effective to migrate the web application to non-blocking web frameworks than to continue horizontal scaling. This research was aimed at elaboration of guidelines for estimating profits[6] of the migration from blocking to non-blocking web frameworks and provides an aid to decide if it is worth to migrate.

The remainder of this document is structured as follows. *Background* (Chapter 2) describes research background and provides introduction into main concepts that are being investigated in the following chapters. *Related work* (Chapter 3) emphasizes on the summary of related works regarding web application performance. *Research methodology* (Chapter 4) defines the research goals and research questions. After that the research design and methodology are discussed. *The maximal performance improvement* (Chapter 5) describes the experiment that was conducted to evaluate the maximal performance improvement. *The real performance improvement* (Chapter 6) describes the experiment that was conducted to evaluate the performance improvement on the existing ROA based web application. All works conducted in the thesis are summarized in *Summary* (Chapter 9). The *Appendix A* contains the detailed experiments results. The *Appendix B* contains the methods and equations that were used for hypothesis evaluation in the experiments. The *Appendix C* contains the server configuration that was used during the experiments.

[4]HttpClient, Http Kit, Apache CXF, RESTEasy, Spray are examples of software that can be named as web application platform, full-stack framework, or micro-framework. As the name is not unified, in this research they are all called web frameworks.

[5]In this research only ROA based web applications, which are composed of distributed REST web services, are being investigated

[6]In this research the profits are seen as advantages of the migration i.e. number of servers that can be saved

Chapter 2

Background

This chapter gives a short introduction into web application requests handling, ROA based web applications and performance characteristics.

2.1 Connections handling

Web application works in such a way that multiple concurrent clients communicate with a server using mutually independent connections. The communication between a client and a server is based on the HTTP protocol. The RFC 2616[1] defines HTTP/1.1, which is the version of HTTP currently in common use.

An HTTP client initiates a request by establishing a Transmission Control Protocol (TCP) connection to a particular port on a server. Then HTTP server is listening on that port for client's requests. When a client sends a request, the server sends back a response. Typically, after that the TCP connection is closed. However, the time needed to establish a new connection is high in compared to the time needed to send a request. To avoid that the keep-alive mechanism is preferred, where a connection can be reused for more than one request.

2.1.1 Blocking and non-blocking web frameworks

After establishing a connection the way that a web application handles the incoming requests affects its ability to scale. A request can be handled in two ways: using blocking or non-blocking I/O operation [2]. In blocking approach when a thread performs an I/O operation, such as writing to a TCP port, it is blocked until the operation is finished. The processor switches its execution context to other threads while awaiting for the blocked thread to be ready again. Non-blocking approach offers more efficient and scalable alternative [5]. It uses event notification facilities to register interest in certain events, such as data being ready to read on a particular port. Instead of waiting for an operation to complete, a thread can work on something else and handle the result of the operation when an event occurs.

[1] https://tools.ietf.org/html/rfc2616

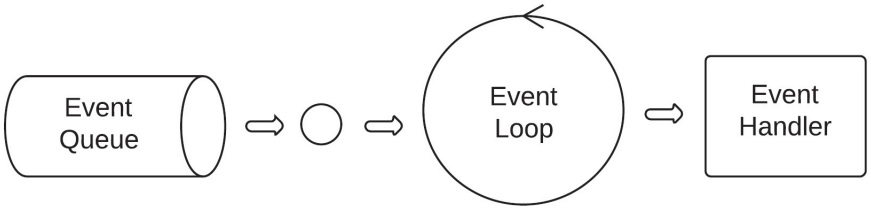

Figure 2.1: A single-threaded event loop consumes events (circles) from the queue and sequentially executes request handles.

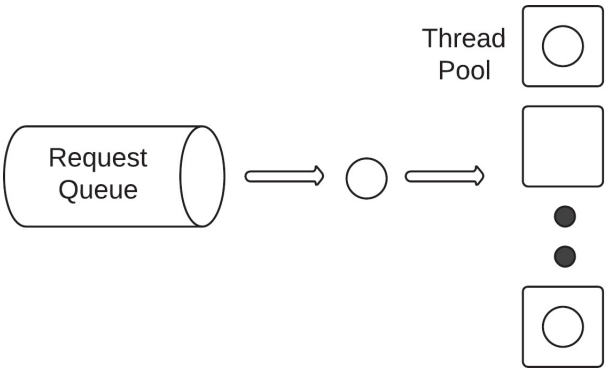

Figure 2.2: A sample thread pool (squares) with requests (circles) that comes from request queue.

2.1.2 Thread pool and event driven strategies

There are two strategies to ensure the parallel requests handling: thread pool and event driven. The first one assigns a different thread for each request and uses blocking I/O operations (Figure 2.2). The size of the thread pool defines the number of requests that can be handled simultaneously. The latter usually uses one thread for request handling and perform I/O operations in non-blocking way (Figure 2.1). In this strategy one thread is capable of handling many requests simultaneously [6].

In thread pool strategy, when the size of the pool is large, overhead due to thread scheduling and context-switching can degrade the performance [4]. Study [6] argue that event driven strategy allows building a Java web application that scales as well as native-compiled web server such as Apache. Another research [4] shows that event-driven based server provides better performance (up to 18%) than a server that uses a thread pool for request handling.

2.1.3 Web frameworks

There are many web frameworks that implement either blocking or non-blocking approach for requests handling. Apache CXF, Jersey, Jetty, RESTEasy, Spray are examples of such web frameworks. The web frameworks differ in the performance, features and design.

NIO is a collection of Java APIs that offers non-blocking I/O operations. NIO consists of buffers, channels and multiplexed, non-blocking I/O facility for writing scalable servers[2]. The select operation on channels is used in the implementation of the server's main loop to detect which web clients have sent requests to the server or which sockets have more data available [6].

In blocking mechanism, a thread cannot do anything more until the I/O is fully received. The application's flow is blocked because the methods do not return right away. Non-blocking mechanisms immediately queue an I/O request and return the control to application flow [7].

Grizzly[3] and Netty[4] are examples of web frameworks that are built on top of NIO that enables quick and easy development of scalable web applications.

2.2 ROA based applications

Representational State Transfer (REST) is an architectural style used for accessing resources on the web [8]. Applications that follows REST style are built in a client-server model, use stateless communication protocol, have an uniform interface for all parts of the system and a layered resource hierarchy. REST is an abstraction of the HTTP Protocol and concentrates on concepts instead of on technical details [9]. REST architectural features and restrictions allow for high scalability and expansion of web applications. Each REST service consists of three components: the URI of the web service, the data type supported (JSON, XML) and the supported operations (HTTP methods).

Scalability is especially important for architectures that compose of distributed services such as Resource Oriented Architecture (ROA). ROA based web applications compose of resources. Each resource is a distributed component that exposes REST based services [10]. Those services communicate with each other to process the requests.

In a distributed architecture, the performance depends on the cost of transmitting data over the network and on the way that single node handles requests [10]. These factors depend on the communication protocol (HTTP) and whether the nodes uses blocking or non-blocking I/O operations.

In a blocking approach, when a node makes a call to another node, the thread

[2]http://www.jcp.org/en/jsr/detail?id=51
[3]https://grizzly.java.net/
[4]http://netty.io/

that handles the request is blocked until the response occurs. In order to handle many requests simultaneously, a high number of threads in a thread pool is required. In a non-blocking approach for handling incoming and outgoing requests, a small number of threads is required as the threads do not have to wait until the outgoing request is completed [5]. Thanks to that is it possible to handle more clients simultaneously, and the response times are relatively smaller. The computation time is not wasted on context switching, as in blocking approach [4].

2.3 Performance characteristics

In related studies [6, 1, 4, 5] an experiment was conducted to measure the performance of web applications. The performance characteristics were measured using HTTP benchmarking tools.

In this paper the performance of a web application is characterized using following attributes:

- RPS - the number of successful requests per second that an web application can handle
- RT - the response time measured from sending a request and receiving a response from a server

In related studies the following HTTP benchmarking tools were used:

- AB[5] - Apache HTTP server benchmarking tool
- HTTPerf[6] - HTTP performance measurement tool developed by Hewlett Packard
- wrk[7] - modern HTTP benchmarking tool capable of generating significant load when run on a single multi-core CPU

The wrk was chosen as a benchmarking tool for the experiments. It is the most modern benchmarking tool and can utilize multi-core CPU fully. It combines a multi-threaded design with scalable event notification systems such as epoll and kqueue. All the requests are sent using non-blocking I/O operations. The other tools uses blocking I/O operation and because of that the results of the performance measurement are less precise.

[5] http://httpd.apache.org/docs/2.4/programs/ab.html
[6] http://www.hpl.hp.com/research/linux/httperf/httperf-man-0.9.txt
[7] https://github.com/wg/wrk

Chapter 3

Related work

In this chapter the most relevant studies contributing to the performance of web applications, blocking and non-blocking solutions and Resource Oriented Architecture are presented. The studies referenced here were mostly collected during the background research.

Arun Iyengar and Daniela Rosu in their research [11] focused on the performance, scalability and availability of web applications. The outcome of the paper is an overview of recent solutions concerning architectures and infrastructures used in building web applications.

In the paper [4] David Pariag et al. compare the performance of web servers based on three different server architectures: event-driven, thread-per-connection and hybrid pipeline-based. They show the importance of proper tuning of the server architecture. The correct combination of the number of connections and worker threads is required. They found that event-based or pipeline-based servers perform as well as or better than the thread-per-connection server up to 18%.

In a study [6] Vicenc Beltran et al. also examine event-driven architecture for building high-performance web applications. They focus on Java platform and evaluate the scalability that NIO API library provides. The conclusion is that this library can be successfully used to create even-driven Java server that can scale well. This solution reduces the complexity of the code and the system resources required to run it. They found that there is no more need to use native-compiled Java Web Servers as web interface for Java Application Servers when these tools are used.

Daniele Bonetta et al. in the paper [5] present a new event-based programming framework for the JVM that allows developing high-throughput web services in Scala. It features an event-based programming model, implicit parallelism and safe state management. When using it developers does not have to deal with processes, threads, locks and barriers. They can focus on business logic. Conducted experiments show that services developed that way yield linear scalability and high throughput when deployed on multicore machines.

U. Gokhale, A. Gokhale and J. Gray in a paper [12] evaluated the performance of an asynchronous web server as a continuation of their previous research [13]. They present the characteristics of Proactor pattern and a queuing model of an

asynchronous Web server implemented using it. They demonstrate the use of the model and provisioning decisions with several examples.

G. Tretola and E. Zimeo in their research [14] focus on asynchronous invocation and continuation. Those patterns are common in middleware infrastructures for distributed computing. They identified abstract interaction pattern and interaction necessary to support asynchrony behaviors. Then they designed and developed two implementations of this pattern and conducted several kinds of testings to access performances. They argue that asynchronous invocations and futures perform always better that synchronous invocation.

Experiments conducted by Jacques M. Bahi et al. in [15] evaluated the behavior of Java iterative applications with the following conditions: one and several sites, a large number of processors (from 80 to 500), different communication protocols (RMI, sockets and NIO), synchronous and asynchronous model. The results show that the asynchronous version of the solver is always faster than the synchronous one and NIO is always faster than the other communication protocols, that were tested (RMI and Socket) and leads to better scalability.

The non-blocking approach was evaluated in practice in [16]. The authors compared modern I/O frameworks by running a series of stress tests. As the results, they selected NIO.2 as the most appropriate framework for JBoss connector.

Sadek Drobi, in an article [17] describes the Play as a modern web application framework. The Play architecture is based on entirely reactive model that focus on the non-blocking invocation. He states that Play's strategy is to put "Web's power in developers' hands, rather than abstracting and concealing its existence".

The NIO is not only useful for web application. Study [18] proved that NIO can improve that performance of Message Passing interface (MPI) for High Performance Computing (HPC) applications. In this paper authors evaluated the performance of legacy Java I/O API and Java NIO API in the HPC context. The results show that NIO provides significant performance gains and performs even better with increasing number of processes.

B. Upadhyaya et al. in the paper [19] evaluated the migration from SOAP-based services to RESTful services. The case studies show that REST services have more performance benefits than SOAP based services. The advantages are simplicity, interface flexibility, interoperability, and scalability. REST resources can easily interact with other web resources.

A research that evaluates the performance of non-blocking I/O in the context of ROA based web applications was not found in the literature. It was determined as a knowledge gap that requires research.

Chapter 4

Research methodology

This chapter describes research methods that are used to achieve the aim and objectives of this study. It starts with problem definition and its motivation (Section 4.1) which is followed by description of the aim, objectives (Section 4.2) and research questions (Section 4.3). Then the methodology overview is presented which maps the research questions to the methods (Section 4.4). In the end, the suitability for each method is given (Section 4.5).

4.1 Purpose and motivation

This research was inspired by real problem stated by the company A. In their recent projects they were using blocking web frameworks for communication between services. They were interested in how the migration to the non-blocking solutions can improve the performance of their recent large scale project. This may be a common problem for many other companies that are also considering migration from blocking to non-blocking web frameworks, but they do not know the possible outcomes, advantages and benefits.

Company A specializes in the architecture and deployment of Identity and Access Management (IDAM) systems. To build IDAM systems, as main components they are using JVM based products (OpenAM, OpenDJ, OpenIDM) that expose REST services. Because of that the overall architecture of systems they build is similar to the ROA. They agreed to provide an ROA based web application that was used as the real web application for testing the migration. Considering this context the research was focused on investigation of ROA based web applications and JVM based web frameworks.

4.2 Aims and objectives

The aim of the thesis project was **to provide guidelines for assessing profits**[1] **of the migration from blocking to non-blocking JVM web frameworks in ROA based web applications.** The objectives to reach the aim were:

[1] i.e. number of servers that can be saved

- to identify the most popular blocking and non-blocking web frameworks
- to evaluate the influence of web framework's type on the performance of ROA based web applications
- to develop guidelines for assessing profits of the migration from blocking to non-blocking JVM web frameworks in ROA based web applications

4.3 Research questions

The following are research questions that were answered in this study to achieve aim and objectives:

- **RQ1:** What are the most popular blocking and non-blocking web frameworks?
- **RQ2:** What is the influence of web framework's type on the performance of ROA based web applications?
  - **RQ2.1:** What is the maximal performance improvement of using non-blocking instead of blocking calls in ROA based web application?
  - **RQ2.2:** What is the performance improvement of using non-blocking instead of blocking calls in a chosen existing ROA based web application?
- **RQ3:** What guidelines for assessing profits of the migration from blocking to non-blocking JVM web frameworks in ROA based web applications can be drawn from the experiments?
  - **RQ3.1:** How to assess the overall performance of ROA based web applications?

4.4 Methodology overview

An overview of the way the research was conducted is presented on figure (4.1). The diagram shows methods that were used to address the research questions, inputs and outcomes. The figure should be read from left to right and from top to bottom.

The aim of the project (**RQ3**) was to provide guidelines for assessing profits of the migration from blocking to non-blocking JVM web frameworks in ROA based web applications. The guidelines were created based on the consultation with the software architects from the company A (**RQ3.1**) and the influence analysis of web framework's type on the performance of ROA based web applications (**RQ2**).

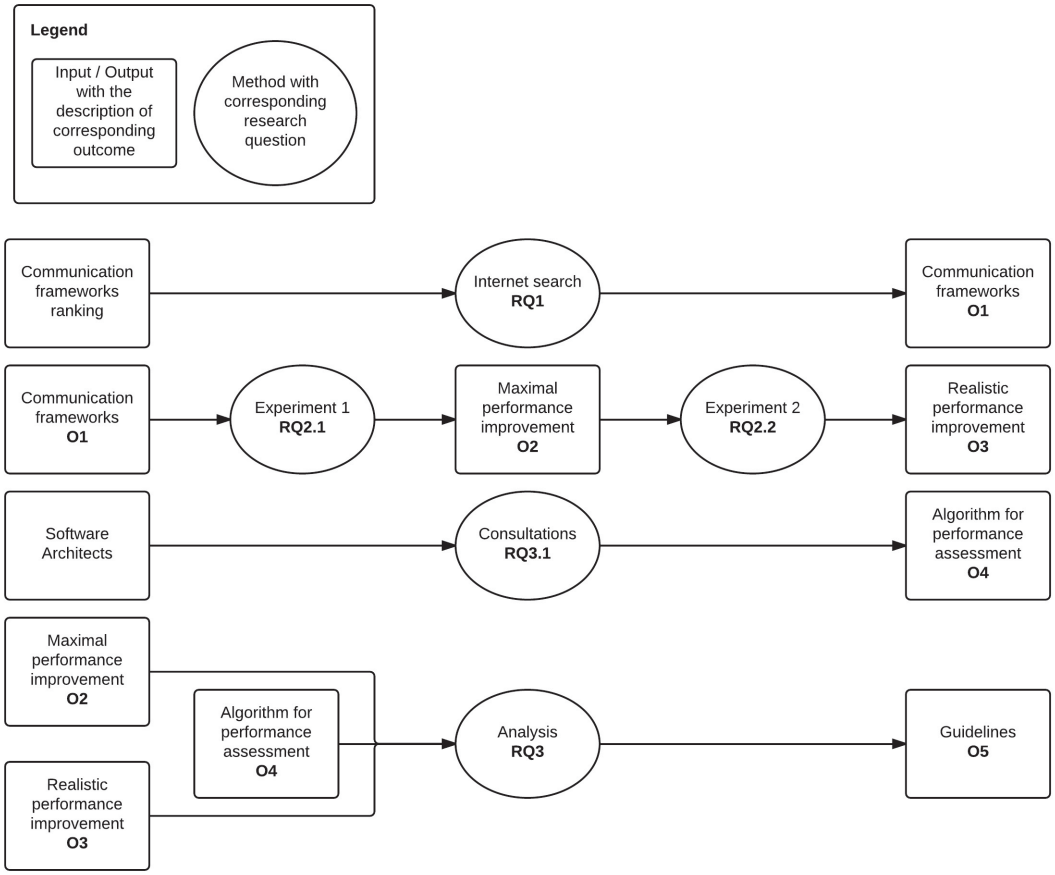

Figure 4.1: Overview of research methods

To evaluate the influence, two experiments were conducted. In the first experiment (**RQ2.1**) the maximum performance of bare[2] web applications, which were built using different web frameworks, was measured. In the second experiment (**RQ2.2**) the performance of sample existing ROA based web application before and after migration was investigated. The sample web application was obtained from the company A, as they are considering the migration and are interested in the result. Internet search was conducted to get a list of popular web frameworks which were used to build bare web applications for the first experiment.

[2]an web application that does not do anything else then accepting requests and returning sample response

4.5 Suitability of selected methodologies

To identify the most popular web frameworks (**RQ1**) a survey could be conducted instead. The survey might identify which web frameworks are currently used in the industry, which would make the whole research more reliable. However, due to lack of resources and time this option was not chosen and was suggested as a future work.

Justification for using experiment as a tool for **RQ2** is based on a fact that experiment was used to measure the performance of a web applications in all the literature related to the analysis of the background. Furthermore, the company A was interested in exact performance improvement evaluation in the provided application, which would not be possible to obtain by conducting a systematic literature review or a survey. In addition, the company A was not interested in performing the performance evaluation by its employees, which eliminated the case study method from the research.

For evaluating the maximal performance improvement (**RQ2.1**) the performance results could be used from the web frameworks ranking[3] instead of performing an experiment. The reason, why it was not used, is that the tests were not performed in an ROA based environment.

To find out how to assess the overall performance of ROA based web application (**RQ3.1**) the survey could be conducted. The survey might provide more general insight on the overall performance evaluation. However, the consultations with the software architects from the company A gives more accurate guides that fit the web application that was evaluated in the (**RQ2.2**). This makes the guidelines more useful for the company A, but less generic. The lack of time was again a reason for not conducting a more broad investigation. The additional survey and interviews were suggested as a future work.

[3] http://www.techempower.com/benchmarks/

Chapter 5

The maximal performance improvement

The aim of this chapter is to address the RQ1, i.e. *What are the most popular blocking and non-blocking web frameworks?* and RQ2.1, i.e. *What is the maximal performance improvement of using non-blocking instead of blocking calls in ROA based web application?*. It describes the experiment that was conducted to evaluate the maximal performance improvement of the migration from blocking to non-blocking web frameworks in an ROA based web application. The experiment was constructed according to "Experimentation in Software Engineering" [20]. This chapter is structured into five parts: scoping (Section 5.1), planning (Section 5.2), operation (Section 5.3), analysis (Section 5.4) and summary (Section 5.5).

5.1 Scoping

This section formulates the problem and defines the objective and goals of the experiment.

5.1.1 Goal definition

The experiment is motivated by the need to evaluate the maximal possible performance improvement that can be gained from the migration from blocking to non-blocking web frameworks in ROA based web application. In order to analyze if the migration is cost-effective, it is essential to know the upper limit of the performance improvement that is not possible to exceed.

Study objects The objects of study are the most popular JVM based blocking and non-blocking web frameworks.

Purpose The purpose of the experiment is to evaluate to maximal performance improvement gained from the migration from blocking to non-blocking web frameworks in an ROA based web application.

Perspective The perspective is from the point of view of the users of an web application.

Quality focus The main criterion is the maximal number of requests per second that a single server can handle while 90% of the response times are below specific value that is acceptable by the visitors of a web application.

Context The context of the experiment are bare web applications that were built using blocking and non-blocking web frameworks.

The aim of the experiment can be defined as follows:

Analyze the *migration from blocking to non-blocking web frameworks*
for the purpose of *the maximal performance improvement evaluation*
with respect to the *number of requests per second*
from the point of view of *the web application users*
in the context of *ROA based web application*

5.2 Planning

This sections describes the experiment design and provides high-level overview on how the experiment was conducted.

5.2.1 Hypothesis formulation

Let:

- RPS_B be the maximal number of requests per second that a single server can handle while the 90% of response times are below 200 ms while hitting REST endpoint implemented using blocking web framework and
- RPS_{NB} be the maximal number of requests per second that a single server can handle while the 90% of response times are below 200 ms while hitting REST endpoint implemented using non-blocking web frameworks

Then, the hypotheses are formulated as follows:

- H_0: $RPS_B = RPS_{NB}$
- H_1: $RPS_B <> RPS_{NB}$

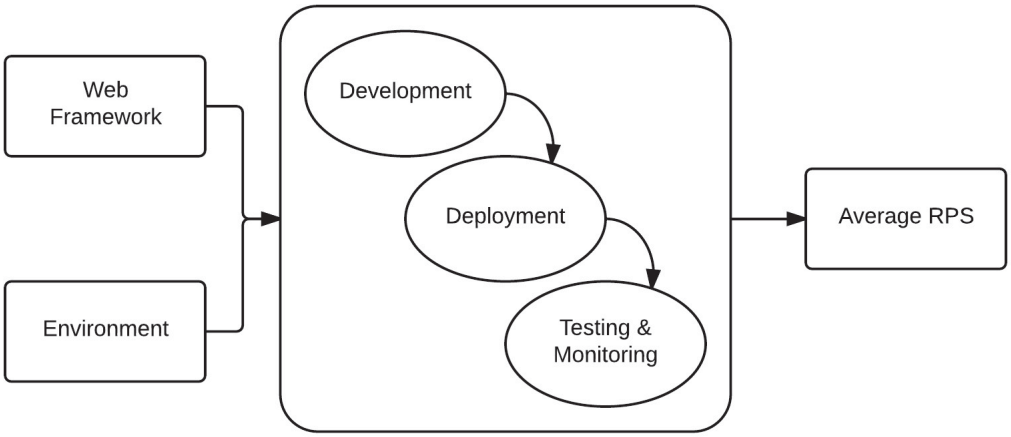

Figure 5.1: Experiment design

5.2.2 Variables and subjects

The subjects are the most popular blocking and non-blocking web frameworks that are used to build bare web applications that expose single REST endpoint.

The independent variables are the web framework used to build the web application and environment on which the tests are run. The dependent variable is the mean number of requests per second that a single server can handle while the 90% of response times are below 200 ms while hitting REST endpoint.

5.2.3 Experiment design and instrumentation

The Figure 5.1 presents the high-level experiment design. The inputs (independent variables) for the experiment are a web framework and an environment. The output (dependent variable) is average number of RPS that an web application implemented using given web framework and run on given environment can handle.

In order to transform input variables into output variable several steps need to be performed. First it is required to build an web application using given web framework. Next a unified way of deploying the web application to the testing environment needs to be developed. Then an instrument for performance testing needs to be prepared. The environment should be monitored during the tests. Following paragraphs describe each step in more detail.

Development Before the execution of the experiment, the most popular web frameworks need to be selected. One of them must be used as the reference web application that expose one REST endpoint which returns an example response.

For each selected web framework an web application needs to be build. Each web application need to expose one REST endpoint which should make a call to the reference web application and pass the response back to the client. That way it will simulate the ROA based web application. The performance depends mainly on the way that web application handles incoming and outgoing requests (blocking or non-blocking).

The web applications should be tuned accordingly to its documentation in order to provide high performance. Furthermore, all the web applications should be tested before running the final tests in order to verify if the implementation is correct and if they behave in similar predicted way.

Deployment To be able to install and configure a server in fast and repeatable way on any environment the unified way of deployment needs to be prepared. The tool need to be able to install developed web application, an instrument for testing and monitoring tools on target server. Furthermore, it should configure and tune the server to provide high performance. Thanks to that it will be possible to repeat the experiment on any other environment in the future.

Testing & monitoring To measure the performance, a benchmarking tool is required. The tool need be able to generate the heavy load by sending a REST calls to the server in parallel and calculate the statistics. It should be possible to configure the number of concurrent connections and the duration of the test. In addition to that, it should be possible to repeat the test many times, and then reject the best and the worst results and calculate the average number of RPS.

While the tests are being run it is necessary to monitor the utilization of the CPU, memory and network. It is also required to verify the number of concurrent connections that were established with the server. Such monitoring tools need to be discovered.

5.2.4 Validity evaluation

There are four levels of validity threats to consider: conclusion validity, internal validity, construct validity and external validity.

Conclusion validity Threats to conclusion validity concern the issues to draw the correct conclusion. One general threat to conclusion validity is that the maximal number of requests per second, while 90% of response time are below some defined limit, may not be appropriate measure to compare the performance of blocking and non-blocking web frameworks. This method emerged from the consultations with the software architects from the company A as this is the approach they use to approximate the performance of theirs systems.

Internal validity Internal validity concerns issues related to subjects, independent variables and instrumentation. The limited number of web frameworks is being tested. Thus, the performance results may depend on the quality of web framework's implementation. Furthermore, the key to obtain the maximal results is to tune the operating system and software properly. However, all web frameworks are being tested on the same environment, so the results should be comparable.

Construct validity Construct validity concerns generalization of the experiment result to the theory behind the experiment. A major threat to the construct validity is that the mean number of requests is used to compare the results. It may affect the conclusions if the results are not filtered properly before calculating the mean value.

External validity External validity concerns generalization of the experiment result to other environments. There are two major threats to external validity. The first one is that only limited set of web frameworks were tested, thus results may be different using other web frameworks. It was not possible to extend the number of web frameworks due to limited amount of time. The second one is that the experiment was not conducted on a dedicated physical server The performance of web web application depends on server resources, network, operation system, proper tuning, web application's server and web frameworks, thus the results may be distinctive on different environments. Although the ratio should be relatively similar.

5.3 Operation

This section provides low-level description on how the experiment was conducted.

5.3.1 Preparation

This section describes all steps that were taken before the execution of the experiment. All the web applications, scripts and tools that were used in the experiment and were created by the author are publicly available at Github[1]. Thanks to that the experiment can be easily repeated on other environments and extended.

Web frameworks The three most popular blocking and non-blocking JVM based web frameworks were selected from the web frameworks ranking[2].

[1] https://github.com/mbilski/benchmark
[2] http://www.techempower.com/benchmarks/

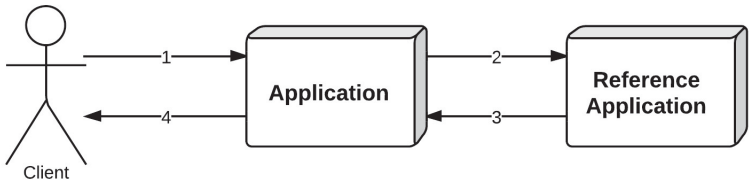

Figure 5.2: Diagram of the web flow

Environments In order to make the results more reliable, the experiment was performed on two environments. The first environment consisted of two laptops connected to each other via Gigabit Ethernet cable. The second environment consisted of a single Amazon AWS virtual machine. The details are presented in Table 5.2. All of them were running on Ubuntu 12.04. The JDK 7 from Oracle was used as Java Runtime Environment.

Initially, the tests were performed on two AWS instances. However, the network limitations affected the results. It was decided to conduct an experiment on only one instance, in order to avoid it.

Development The selected web frameworks were used to build six bare web applications. All the prepared web applications exposed one REST endpoint which makes a call to the reference web application and passes the response back to the client. The Figure (5.2) shows the web flow between the client, web application and reference web application. The reference web application is the same for all web applications and exposes one REST endpoint which responses with simple JSON object. For each web framework, in order to make the call from web application to the reference web application, the REST client recommended in the documentation was used. The list of web application with corresponding client and container is presented in Table 5.1. Web application that was built using Spray web framework was used as a reference web application.

All the web applications were initially tested, in the form of a pilot experiment, to verify the correctness of the implementation and configuration. It was confirmed that all the web applications provided comparable performance. None of the web application diverged from the other web applications of the same type in terms of the performance.

Deployment Chef is a configuration management tool written in Ruby. Chef was used to automate the process of configuration the environment on which the tests are run. In addition, Chef was used to install all the required tools and tuned the server to provide high performance. Thanks to Chef it is possible to prepare environment for testing within few minutes using only one command.

Table 5.1: Web applications

Framework	Container	Client Framework	Type
Spray 1.3	spray-can 1.3	spray-client 1.3	non-blocking
Grizzly 2	grizzly-http 2.3.14	async-http-client 1.8.12	non-blocking
Servlet 3.0	jetty-servlet 9.1.0	jetty-client 9.1.0	non-blocking
Resteasy 3.0.7	Tomcat 7.0.34	resteasy-client 3.0.7	blocking
CXF 3.0.0	Tomcat 7.0.34	cxf-rt-rs-client 3.0.0	blocking
Jersey 2.10.1	Tomcat 7.0.34	http-client 4.3.4	blocking

Table 5.2: Environments

Environment	Instances	CPU	vCPU	Memory	Network
Local	2	i7 2.4 GHz	8	8 GB	1 GigE
AWS m3.xlarge	1	Xeon E5-2670	4	15 GB	-

Testing The wrk was used as an HTTP benchmarking tool (based on the background research conducted in the Section 2.3). A Ruby script was prepared which allowed to run the benchmark several times and return the mean number of requests per second and mean latency. Before calculating the mean the best and the worst results were removed from the set. Table 5.3 shows the configuration options of the script.

During the pilot experiment this tool was used to verify the implementation of prepared web applications. In addition, as a result of the pilot experiment, the parameters of the tool (number of connections, duration, number of repeats) were established.

Monitoring JVisualVM was used to monitor JVM processes. It allowed to check CPU & memory usage and how many threads were used. System monitor was used to monitor overall CPU, memory and network usage. Netstat was used to validate how many connections were established on a given port.

5.3.2 Execution

Deployment Chef and scripts were used to install on the server all required tools and prepare the web applications. Before running the web applications the number of maximal opened ports was increased to 64000 using ulimit and sysctrl commands. Thanks to that it was possible to measure the RPS while 50000 simultaneous connections were established by the load generator. Each web application was configured to use maximum of 2048 MB from available memory. The full specification of server configuration can be found at Appendix C.

Table 5.3: Testing tool options

Attribute	Description	Default
url	URL of web application's endpoint	http://localhost:8080
threads	Number of threads to use	8
duration	Duration of single test	30s
connections	Connections to keep open	1000, 10000
repeats	Number of repeats	5
delay	Delay between tests	5s

Testing The benchmark tool was run for 1000, 10000, 20000, 30000, 40000 and 50000 simultaneous connections for each prepared web application. Single test lasted 30 seconds. For each number of connections the test was repeated 5 times. The best and the worst results were rejected. There was a 10 seconds delay between each repeat. On Amazon AWS instance, the tests were run only up to 20000 connection due to the the limited amount of available ports on a single instance.

Monitoring The CPU, memory and network usage was monitored during the tests. The CPU usage varied from 50% to 80% depending on the web framework's type. Blocking web frameworks used to have higher CPU usage than non-blocking web frameworks. The memory usage never exceeded the limit. The network usage varied from 3 MB/s to 12 MB/s. The network usage and requests per second were mutually depended. If network usage was high, then the test result was high too. The netstat command was used to validate if required number of connections were established on given port. Usually it took a few seconds at the beginning to achieve specified level of connections.

Results Figure (5.3) shows requests per second for different number of simultaneous connections for blocking and non-blocking web frameworks on i7 environment. Figure (5.4) shows corresponding mean latency for requests per second from previous plot. Figures (5.5) and (5.6) show requests per second and mean latency from running the tests on Amazon AWS instance. More detailed results can be found in Appendix A.

On all figures which show requests per second the non-blocking web frameworks have higher performance. In addition, from the analysis of the figures that show latency it can be said that the non-blocking web frameworks have higher scalability. The latency grows slower for non-blocking web framework with increasing number of concurrent connections than for blocking web frameworks.

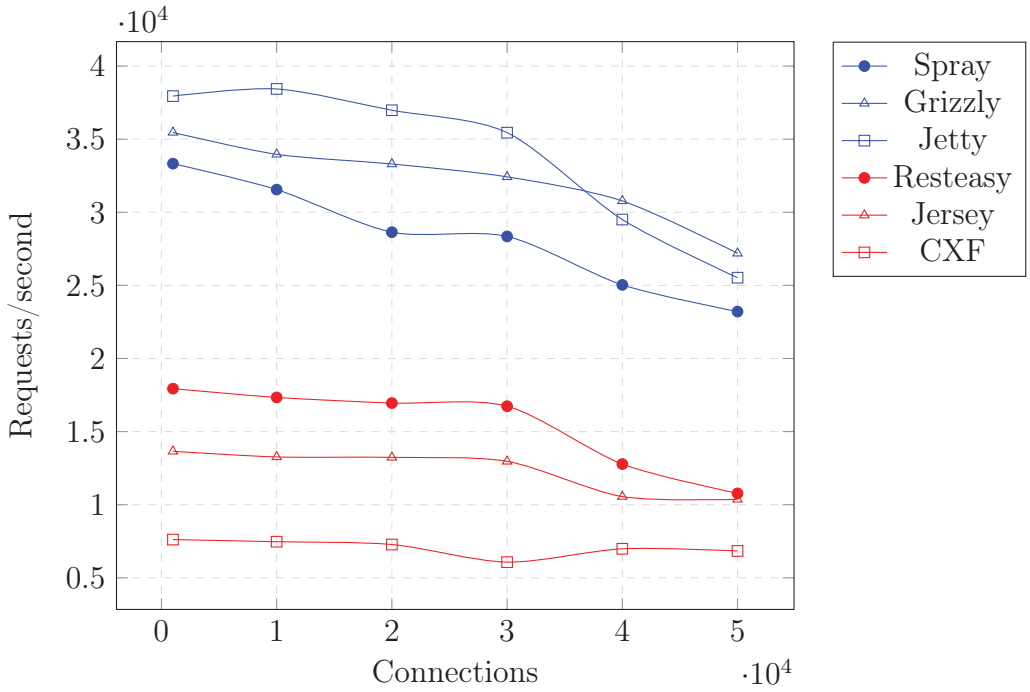

Figure 5.3: RPS on i7

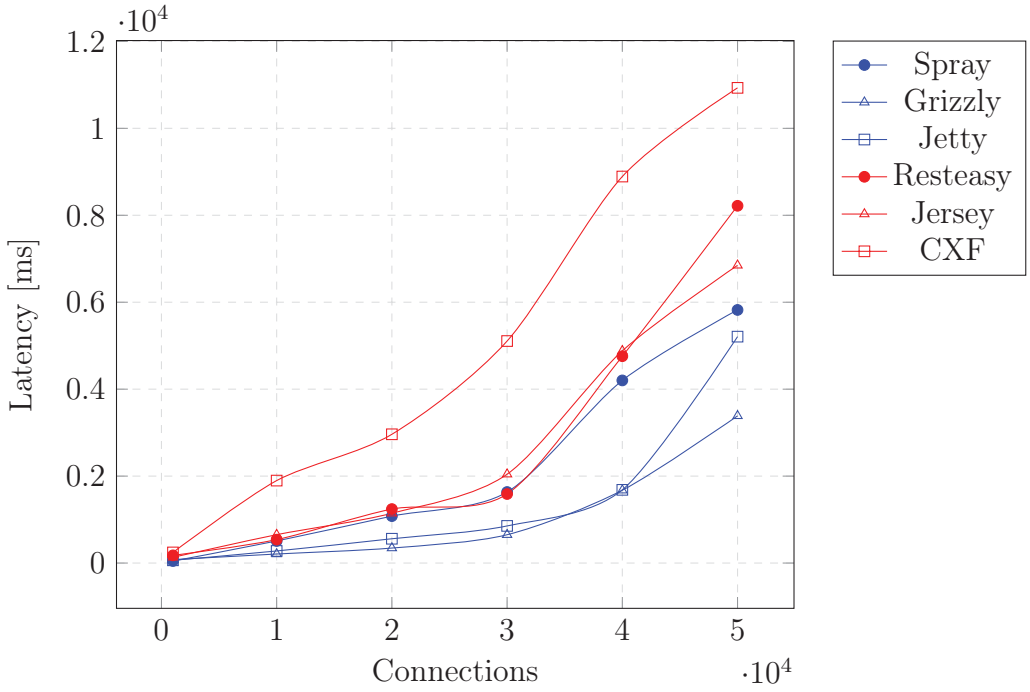

Figure 5.4: Latency on i7

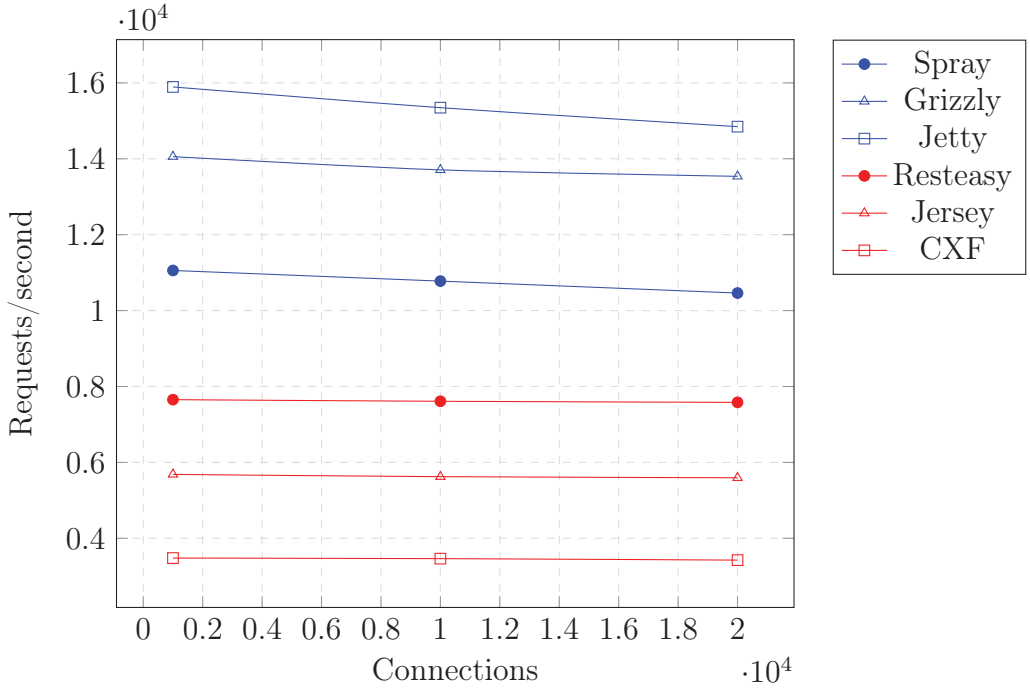

Figure 5.5: RPS on Amazon AWS

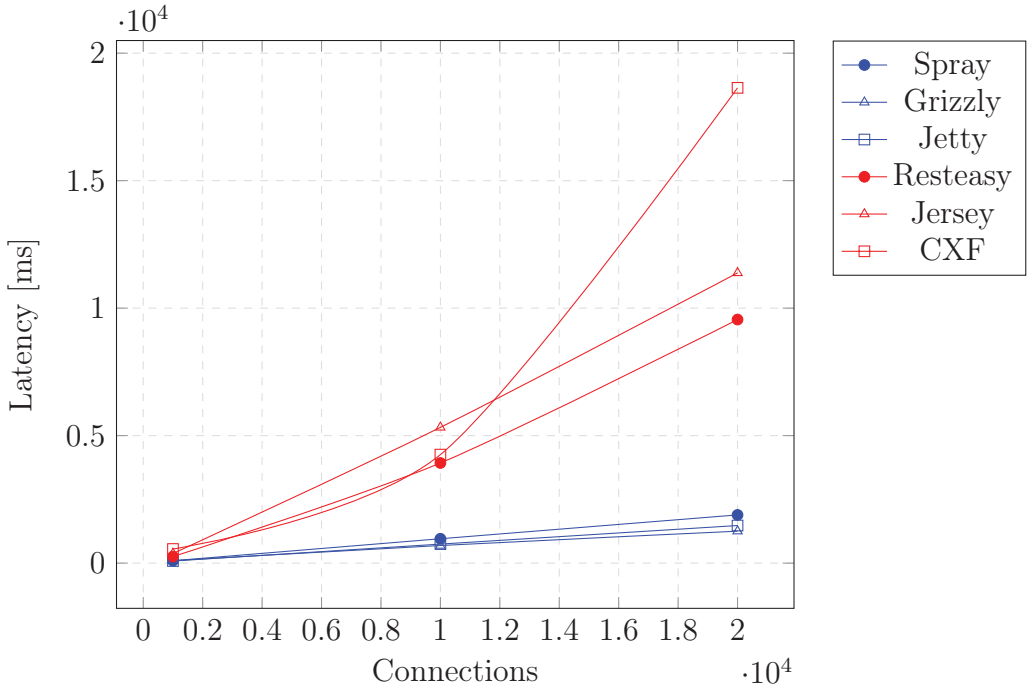

Figure 5.6: Latency on Amazon AWS

Table 5.4: Results from the t-test

Factor	Degrees of freedom	S_p	t_0
B vs NB i7	4	4014	-6.87
B vs NB AWS	4	2271	-4.35

Table 5.5: i7 statistics

Type	Mean value of RPS	Standard deviation of RPS	Mean value of latency [ms]	Standard deviation of latency [ms]
B	13069	5186	184	56
NB	35578	2312	63	16

5.4 Analysis

This section presents the statistical analysis of the gathered data.

5.4.1 Statistics

The mean values and standard deviations of requests per second and latency were calculated for i7 and AWS environments. The results for 1000 simultaneous connections were only taken into the consideration, as for other numbers of connections the latency exceeded the limit of 200 ms stated in the hypothesis. Table 5.5 presents statistics for i7 environment. Table 5.6 presents statistics for AWS environment. On both environments the performance of non-blocking web frameworks was about 2.5 times higher then blocking web frameworks.

5.4.2 Hypothesis testing

The hypothesis was evaluated using a t-test [20]. The results from the t-test are shown in Table 5.4. The critical value for two-tailed t-test (5%), when degree of freedom equals 4 is 2.776. Since $|t_0| > t_{0.025,4}$ in both i7 and AWS cases it is possible to reject the null hypothesis. More details about t-test can be found in Appendix B.

By rejecting the null hypothesis the alternative hypothesis, that mean of the maximal number of requests per second is different when using blocking and non-blocking web frameworks, was proved. Moreover, it also revealed, as the secondary result, that non-blocking web frameworks provide performance up to 2.5 times higher than blocking web frameworks.

Table 5.6: Amazon AWS statistics

Type	Mean value of RPS	Standard deviation of RPS	Mean value of latency [ms]	Standard deviation of latency [ms]
B	5605	2088	395	149
NB	13669	2441	91	8

5.5 Summary

The experiment showed that maximal number of requests per second that a single server can handle is higher when using non-blocking then blocking web web frameworks in ROA based web application.

The performance and quality of implementation of each selected web framework, which was qualified as independent variable, had a significant influence on the overall results. Thus, it is not possible to give a precise value of maximal performance improvement that can be gained from the migration from blocking to non-blocking web frameworks in ROA based web application. However, from the analysis of the results from i7 and AWS environments the improvement can be approximated. It can be said that migration increases the number of RPS up to 2.5 times. This approximation can encourage developers and architects of ROA based web application to consider such migration and evaluate it in their environment.

Chapter 6
The real performance improvement

The aim of this chapter is to address the RQ2.2, i.e. *What is the performance improvement of using non-blocking instead of blocking calls in a chosen existing ROA based web application?*. It describes the experiment that was conducted to evaluate the performance improvement of the migration from blocking to non-blocking web framework in the existing ROA based web application. The experiment was constructed according to "Experimentation in Software Engineering" [20]. This chapter is structured into five parts: scoping (Section 6.1), planning (Section 6.2), operation (Section 6.3), analysis (Section 6.4) and summary (Section 6.5)

6.1 Scoping

This section formulates the problem and defines the objective and goals of the experiment.

6.1.1 Goal definition

The experiment is motivated by the need to evaluate the real performance improvement that can be gained from the migration from blocking to non-blocking web frameworks in an existing ROA based web application.

Object study The object of study is a sample migration from blocking to non-blocking web framework in an existing ROA based web application.

Purpose The purpose of the experiment is to evaluate to performance improvement gained from the migration from blocking to non-blocking web framework.

Perspective The perspective is from the point of view of the users of the web application.

Quality focus The main criterion is the maximal number of requests per second that a single server can handle, while 90% of the response times are below a specific value, without a failure.

Context The context of the experiment is the existing web application developed by the company A (Section 4.1).

The aim of the experiment can be defined as follows:

Analyze the *migration from blocking to non-blocking web framework*
for the purpose of *the performance improvement evaluation*
with respect to the *number of requests per second*
from the point of view of *the web application users*
in the context of *the existing ROA based web application*

6.2 Planning

This section describes the experiment design and provides high-level overview on how the experiment was conducted.

6.2.1 Hypothesis formulation

Let:

- RPS_B be the maximal number of requests per second that a single server can handle while the 90% of response times are below 200 ms while hitting REST endpoints implemented using blocking web frameworks in an existing ROA based web application and
- RPS_{NB} be the maximal number of requests per second that a single server can handle while the 90% of response times are below 200 ms while hitting REST endpoints implemented using non-blocking web frameworks in an existing ROA based web application

Then, the hypotheses are formulated as follows:

- H_0: $RPS_B = RPS_{NB}$
- H_1: $RPS_B <> RPS_{NB}$

6.2.2 Variables and subject

The subject is the existing ROA based web application created by the company A that uses blocking calls between components.

The independent variables are the example blocking ROA based web application, a non-blocking web framework used for the migration and environment on which the tests are run. The dependent variable is average RPS improvement gained from the migration.

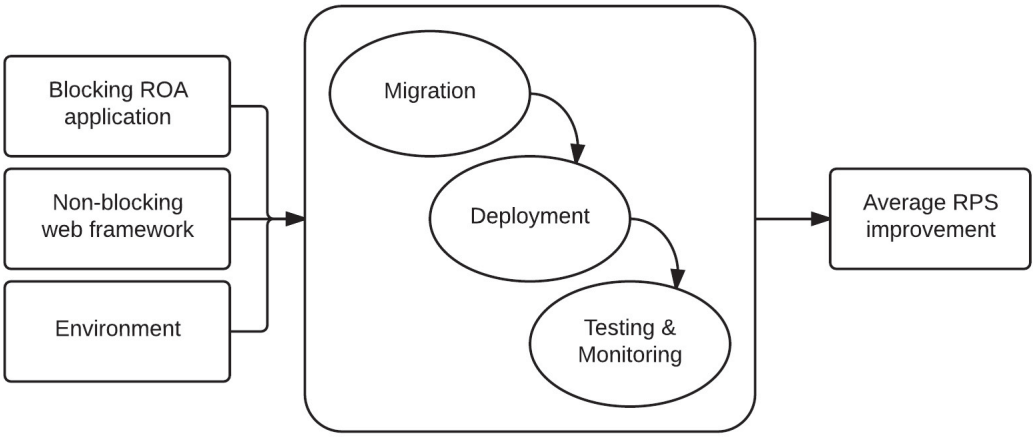

Figure 6.1: Experiment design

6.2.3 Experiment design and instrumentation

The Figure 6.1 presents the high-level experiment design. The inputs (independent variables) for the experiment are blocking ROA web application, non-blocking web framework and environment. The output (dependent variable) is average RPS improvement after migration from blocking to non-blocking web framework for a given web application on a given environment.

In order to conduct the experiment it is required to migrate the sample web application from blocking to non-blocking web framework. First a web framework for the migration needs to be selected. The web framework should fit into web application design and work in the non-blocking mode. In order to accomplish this experiment it is allowed to migrate and test only a limited number of APIs. Thanks to that the process of the migration is simplified.

Then the both versions of the web application should be deployed on the target environment. In the end, both web applications have to be tested. While running the tests, the environment needs to be monitored.

The experiment should be performed on the same environments as in the previous experiment (Section 5.3.1). In addition, the same benchmark and monitoring tools need to be used. Thanks to that the result will be more comparable.

6.2.4 Validity evaluation

There are four levels of validity threats to consider: conclusion validity, internal validity, construct validity and external validity.

Conclusion validity Threats to conclusion validity concern the issues to draw the correct conclusion. One general threat to conclusion validity is that the maximal number of requests per second while 90% of response time are below some defined limit may not be appropriate measure to compare the performance of ROA based web application. However, this is the method that software architects from the company A use to approximate the performance for provided web application. In order to ensure the consistency, the method was not changed.

Internal validity Internal validity concerns issues related to subjects, independent variables and instrumentation. The results depend not only on the performance of the selected web frameworks, but also on the performance and quality of the implementation of the ROA based web application. For any other existing web application the results will be different. However, the observations and conclusions should be equivalent.

Construct validity Construct validity concerns generalization of the experiment result to the theory behind the experiment. The major threat to the construct validity is that the average number of requests is used to compare the results. It may affect the conclusions if the results are not filtered properly before calculating the average value.

External validity External validity concerns generalization of the experiment result to other environments. There are two major threats to external validity. The first one is that the existing ROA based web application was migrated to non-blocking web framework only once. Thus, results depend on the performance of this particular web framework. Due to the limited amount of time, it was not possible to repeat the tests using different web frameworks. The second one is that the experiment was not conducted on a dedicated physical server. The performance of web application depends on server resources, network, operation system, proper tuning, web application server and web frameworks, thus the results may be distinctive on different environments. Although the ratio should be relatively similar.

6.3 Operation

This section provides low-level description on how the experiment was conducted.

6.3.1 Preparation

This section describes the web application, scripts, environments and tools that were used in the experiment.

Migration The web application provided by the company A consisted of three components. The first component was OpenDJ that acted as an LDAP based database. Second component was OpenAM which was responsible for access management. The third component was a web application that exposed Identity and Access Management REST endpoints. Those endpoints made call to the OpenDJ and OpenAM to process the requests.

The Jetty web framework, tested in the previous experiment, was chosen for the migration. This web framework fitted the best the design of the web application and met all the requirements.

Before the migration, the third component was running on Tomcat. It used Resteasy to expose the REST endpoints, Apache's http-client to make calls to the OpenAM and UnboundID as an LDAP client. All requests were blocking. In order to migrate to non-blocking solution, the container and HTTP client were replaced. After the migration, web application was running on Jetty embedded container and was using http-client in non-blocking mode. The client for OpenDJ was not changed.

Deployment To automate the process of setting up the environment the Chef scripts, provided by the company A together with the sample web application, were used. In addition, the scripts from the previous experiment (Section 5.3.1) were used to install the benchmarking tool and tune the server to provide high performance.

Environment and testing In order to make the results from this experiment comparable with the result from the previous experiment (Section 5) the tests were performed in exactly the same environment as previously and using the same benchmarking tool.

Monitoring JVisualVM was used to monitor JVM processes. It allowed checking CPU & memory usage and how many threads were used. System monitor was used to monitor overall CPU, memory and network usage. Netstat was used to validate how many connections were established on a given port.

6.3.2 Execution

Table 6.1 presents the REST APIs that were selected for tests. The web application was deployed to the servers using Chef scripts provided by the company. The Chef scripts allowed for fast deployment and ensured proper tuning of web applications. For all components the logging and cache was disabled. The tests were executed four times: before and after the migration on two environments.

The same settings for the benchmark tool were used as in the previous experiment (Section 5). However, the number of requests per second and latency was

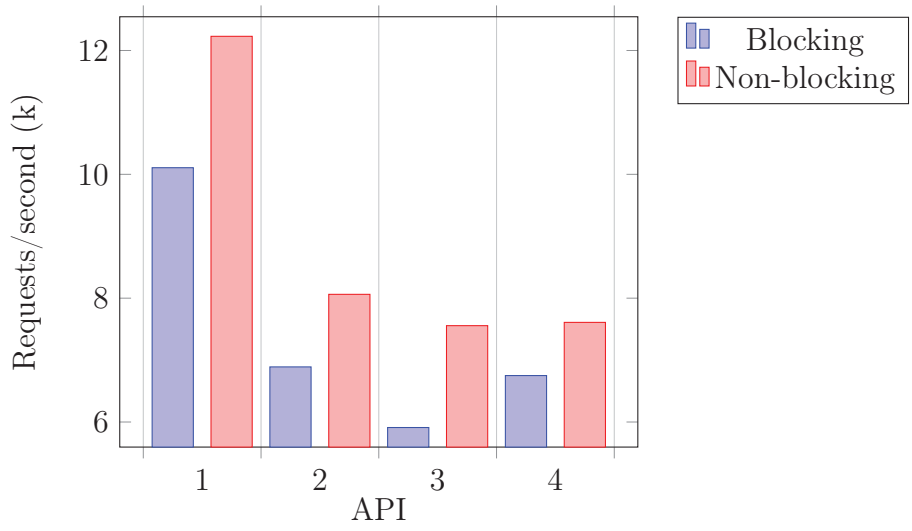

Figure 6.2: RPS on i7

Table 6.1: Tested APIs

N	API
1	Validate Session
2	Get User
3	Get User By Identifier
4	Get Active Sessions

measured only for single value of connections (100) for all APIs.

The figures (6.2) and (6.3) shows the results of benchmark on i7 and AWS environments before and after the migration. Table 6.1 can be used to map the APIs. More detailed results can be found in Apependix A.

During the tests, the CPU usage increased up to 95%. None of the components exceeded the JAVA memory limit. The network usage was lower than in the previous experiment.

6.4 Analysis

This section presents the statistical analysis of the gathered data.

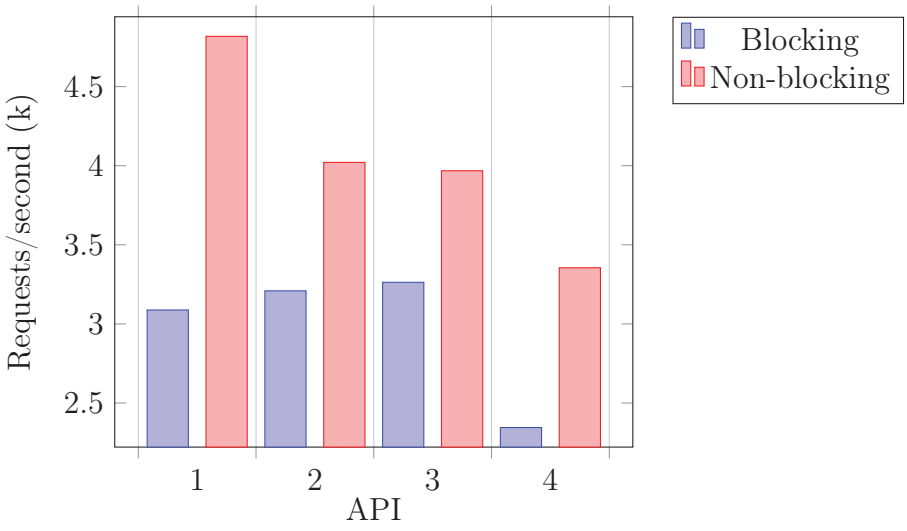

Figure 6.3: RPS on AWS

Table 6.2: Performance improvement

N	API	i7 RPS [%]	AWS RPS [%]
1	Validate Session	21	40
2	Get User	17	25
3	Get User By Identifier	28	22
4	Get Active Sessions	18	43

6.4.1 Statistics

The number of requests per second for each API after and before the migration were used to calculate the improvement. Table 6.2 presents the improvement gained from the migration on i7 and AWS environments.

6.4.2 Hypothesis testing

The hypothesis was evaluated using paired t-test [20]. The paired t-test is useful when two samples resulting from repeated measures are compared. More details about paired t-test can be found in Appendix B.

The blocking and non-blocking value of RPS for APIs on both environments were paired together. The results from the paired t-test are shown in Table 6.3. The critical value for the t-test is 2.365. Since $|t_0| > t_{0.025,7}$ it was possible to reject the null hypothesis. Moreover, it also revealed that non-blocking web frameworks increased the performance of the example web application about 27%.

Table 6.3: Results from the paired t-test

$\bar{d}$	S_d	t_0
-1237.13	459.41	-7.62

6.5 Summary

The experiment showed again that the maximal number of requests per second that a single server can handle is higher when using non-blocking then blocking web framework in existing ROA based web application. The average improvement of the tested APIs equaled 27%. The migration did not require reimplementation of the APIs. Only the HTTP container and HTTP client required the migration.

It is not guaranteed that the same performance improvement will occur for other ROA based web application. The overall performance depends on the web frameworks, quality of the implementation, hardware, network and the performance of each ROA component. However, the results can encourage developers and architects to test the migration in their environment.

Chapter 7

Guidelines

This chapter describes the guidelines for assessing profits of the migration from blocking to non-blocking JVM web frameworks in ROA based web applications. The guidelines were prepared based on the previous experiments and consultations with the software architects of the company A. This chapter is structured into three parts: preparation (Section 7.1), guidelines (Section 7.2) and example usage of the guidelines (Section 7.3).

7.1 Preparation of the guidelines

In order to assess if the migration from blocking to non-blocking web framework is cost efficient, it is required to assess the overall performance of an web application before and after migration. The process of assessing the performance was discussed with the software architects from the company A. Their algorithm based on the performance analysis of APIs on a single server. Next, the results are used to assess the performance of the entire web application which most often compose of many servers. In the end, the assessments are compared with the requirements.

The guidelines were prepared based on those consultations. The guidelines focus on performance comparison before and after migration. In addition, they contain the analysis of how many servers need to be added in case when migration is not sufficient to meet the requirements.

The Figure 7.1 presents the guidelines overview. The algorithm requires input variables and produces output variables. The performance improvements obtained from the previous experiments are used as parameters for the algorithm. Thus, the guidelines approximates the performance improvement in two aspects: maximal and realistic.

The parameter for realistic performance improvement was extracted from the second experiment and depends on the evaluated sample web application. In order to make the results more precise and reliable it is recommended to repeat the second experiment on target web application and use the result as the realistic performance improvement parameter.

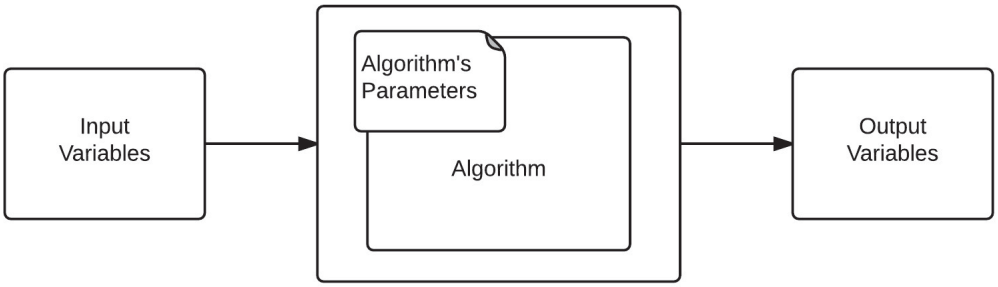

Figure 7.1: Guidelines overview

7.2 Performance improvement analysis

This section describes the process of assessing the maximal and presumable performance improvement that can be gained from the migration from blocking to non-blocking web frameworks in an ROA based web application. Following subsections describes algorithm and input, output variables for the algorithm.

This short glossary explains expressions that are used in the next sections:

- Representational state transfer (REST) - is an architectural style for development of web services
- Application Programming Interface (API) - REST endpoint that provides a service
- Response Times (RT) - is a time between sending a request and receiving a response
- Requests Per Second (RPS) - the maximal number of requests per second that a server can handle, while 90% of response times are below some specific limit, without a failure.

For more detailed descriptions please see the main Glossary at the beginning of this thesis.

7.2.1 Inputs

In order to analyze if the migration from blocking to non-blocking web frameworks is cost-effective, it is required to prepare the list of input variables (Table 7.1). Input variables describes the current state of an web application and requirements. To be able to analyze performance improvement the values that were extracted from experiments are used as parameters for the algorithm (Table 7.3).

Table 7.1: Input variables

Variable	Description
N	the current number of servers
K	the number of REST endpoints that require optimization $i \in [1, K]$
API_i	the list of REST endpoints that require optimization
RT_i	the acceptable maximal response time for API_i
$ACRPS_i$	the current maximal number of requests per second that API_i can handle with 90% of requests below RT_i
$ARRPS_i$	the required maximal number of requests per second that API_i can handle on 90% of requests below RT_i

Table 7.2: Algorithm's parameters

Variable	Description
RPS_{max} [%]	the maximal possible performance improvement
RPS_{real} [%]	the realistic performance improvement

7.2.2 Algorithm

The algorithm for assessing performance improvement of the migration from blocking to non-blocking web frameworks in ROA based web applications contains following steps:

1. Calculate the performance improvements for the whole system
   (a) Calculate maximal RPS for each API_i after migration of the whole system
   $$AMRPS_i = ACRPS_i * RPS_{max} \tag{7.1}$$
   (b) Calculate the prediction for real RPS improvement for each API_i after migration of the whole system
   $$APRPS_i = ACRPS_i * RPS_{real} \tag{7.2}$$
2. Calculate the performance improvements for a single server
   (a) Calculate current RPS for each API_i
   $$CRPS_i = ACRPS_i/N \tag{7.3}$$

Table 7.3: Output variables

Variable	Description
	$i \in [1, K]$
$AMRPS_i$	the maximal RPS for each API_i after the migration
$APRPS_i$	the predicted RPS for each API_i after the migration
$SRPS_i$	the number of servers that needs to be added to the system without the migration to meet the requirements
$WMRPS_i, WPRPS_i$	the number of servers that needs to be added to the system after the migration to meet the requirements (maximal, predicted)

(b) Calculate the prediction for real RPS improvement for each API

$$PRPS_i = CPTS_i * RPS_{real} \tag{7.4}$$

(c) Calculate the maximal RPS improvement for each API_i on a single server

$$MRPS_i = CPTS_i * RPS_{max} \tag{7.5}$$

3. Calculate how many servers need to be added to meet the requirements without the migration for each API_i

$$SRPS_i = \lceil (ARRPS_i - ACRPS_i)/CRPS_i \rceil \tag{7.6}$$

4. Calculate how many servers need to be added to meet the requirements after the migration.

$$\begin{aligned} WMRPS_i &= \lceil (ARRPS_i - AMRPS_i)/MRPS_i \rceil \\ WPRPS_i &= \lceil (ARRPS_i - APRPS_i)/PRPS_i \rceil \end{aligned} \tag{7.7}$$

7.2.3 Output

The application of the algorithm on input variables produces outcomes described in Table 7.3. The algorithm calculates how many servers need to be added to meet the requirements without and with the migration. The number of required new servers after the migration is considered in the maximal and the realistic aspects.

The last step is to compare what is more cost-effective: the cost the migration or the cost of buying new servers. The actual performance improvement should never exceed the maximal value. For most of the APIs the performance improvement will be close to the realistic value.

Table 7.4: Example input variables

Variable	Value
N	4
K	2
	$i \in [1, 2]$
API_i	API_1, API_2
RT_i	200 ms, 200 ms
$ACRPS_i$	40428, 27560
$ARRPS_i$	80000, 60000

Table 7.5: Example parameters

Variable	Value
RPS_{max} [%]	250%
RPS_{real} [%]	27%

7.3 Example usage of the guidelines

This section presents an example usage of the guidelines. The guidelines were applied on an example ROA based web application provided by the company A

Example input variables The input variables for the algorithm are shown in Table 7.4. Table 7.5 presents the percentage improvement values that were extracted from the experiments. Two APIs are being evaluated. API_1 performs operations only on memory, while API_2 reads data from the database. Thus, the initial performance of API_1 is higher.

Example usage of the algorithm

1. Performance improvements for the whole system

$$\begin{aligned} AMRPS_1 &= 40428 * 2.50 = 101070 \\ APRPS_1 &= 40428 * 1.27 = 51344 \end{aligned} \tag{7.8}$$

$$\begin{aligned} AMRPS_2 &= 27560 * 2.50 = 68900 \\ APRPS_2 &= 27560 * 1.27 = 35001 \end{aligned} \tag{7.9}$$

2. Performance improvements for a single server

$$\begin{aligned} CRPS_1 &= 40428/4 = 10107 \\ MRPS_1 &= 10107 * 2.50 = 25268 \\ PRPS_1 &= 10107 * 1.27 = 12836 \end{aligned} \tag{7.10}$$

$$\begin{aligned} CRPS_2 &= 27560/4 = 6890 \\ MRPS_2 &= 6890 * 2.50 = 17225 \\ PRPS_2 &= 6890 * 1.27 = 8750 \end{aligned} \tag{7.11}$$

Table 7.6: Example output values

Variables	Values
$AMRPS_1, AMRPS_2$	97027, 48514
$APRPS_1, APRPS_2$	66144, 33072
$SRPS_1, SRPS_2$	4, 5
$WMRPS_1, WMRPS_2$	0, 1
$PMRPS_1, WPRPS_2$	1, 3

3. Number of servers that need to be added to meet the requirements without the migration.

$$\begin{aligned} SRPS_1 &= \lceil (80000 - 40428)/10107 \rceil = 4 \\ SRPS_2 &= \lceil (60000 - 27560)/6890 \rceil = 5 \end{aligned} \tag{7.12}$$

4. Number of servers that need to be added to meet the requirements after the migration.

$$\begin{aligned} WMRPS_1 &= \lceil (80000 - 101070)/25268 \rceil = 0 \\ WMRPS_2 &= \lceil (60000 - 51344)/17225 \rceil = 1 \end{aligned} \tag{7.13}$$

$$\begin{aligned} WPRPS_2 &= \lceil (80000 - 68900)/12836 \rceil = 1 \\ WPRPS_2 &= \lceil (60000 - 35001)/8750 \rceil = 3 \end{aligned} \tag{7.14}$$

Example output analysis The example output values are presented in Table 7.6. To achieve the requirements without the migration it is necessary to add five new servers. In the best scenario, when performance improvement equals the maximal performance improvement, only one new server will be required. However, the realistic performance improvement says that even after migration it is required to add three more servers. Finally, thanks to the migration at least two servers fewer will be required to meet the requirements.

However, the number of saved servers can be even higher. From the previous experiments, it is known that non-blocking web frameworks improve the performance, because the time wasted on context switching is reduced. For APIs that works really fast and are called often the overall improvement is greater than for complex and slow APIs that are called rarely.

Another factor worth to consider is the cost of the migration. For the example web application used in the second experiment only the HTTP container and client were replaced. It is recommended to compare the cost of the migration with the cost of new servers before making the final decision.

7.4 Summary

The guidelines approximate the number of servers that need to be added to meet the requirements without and with the migration. The maximal and realistic performance improvement obtained from the experiments are used to calculate the upper and lower bounds. The guidelines provide the information how many servers need to be added in the worst and best scenarios. For the example web application the guidelines approximated that at least one server will be saved thanks to the migration.

The prepared guidelines are the fast method for assessing profits of the migration from blocking to non-blocking web frameworks and can be used by product owners or software architects. In order to make the results more precise, it is recommended to repeat the second experiment on target web application. Conducting a second experiment on target web application might provide more accurate results.

Chapter 8

Discussion

The prepared guidelines use the maximal and realistic performance improvements, obtained from the experiments, as parameters for the algorithm. The result of the first experiment can be qualified as the upper boundary of the performance improvement gained from the migration from blocking to non-blocking web frameworks. In theory, there is no real web application that can achieve higher performance improvement gained from such migration. The result of the second experiment, realistic performance improvement, depends on the existing web application that was migrated and evaluated. Because of that, the correctness of the approximation depends on the realistic improvement parameter. The second experiment should be repeated on a target web application to make the approximation more precise.

The example application provided by the company A was stateless and had linear scalability. Thus, the algorithm proposed in the guidelines can be only used on other applications that also behave in a similar way. Future work is required to allow guidelines to approximate performance of applications that does not have linear scalability.

The experiments evaluated the performance of single APIs. It means that the application was handling only one type of requests during the test. In a real scenario, many users call different APIs at the same time. The performance improvement in a real scenario might be different than the assessments obtained from the guidelines.

The results from the first experiment (Chapter 5) show that in theory the performance of non-blocking web frameworks is up to 2.5 times higher that blocking web frameworks in ROA based web application. In blocking approach, two threads are blocked mutually while processing a single request (the first the while handling incoming request, the second while handling outgoing request). Because of that the time wasted on context-switching degrades the performance. However, the second experiment (Chapter 6) show that the performance improvement after the migration equals *only* 27% for the sample existing web application. It shows that the way a server handles the requests does not have such influence on performance for the real web application as for the bare web application. The bare web application's only task was to make a call to another server and send

the response back to the client. Although it is proved that there is a statistical evidence that migration from blocking to non-blocking web frameworks improves the performance.

The modern web frameworks (Grizzly, Netty, Jetty 9), that based on NIO API, supports only non-blocking mode of handling requests. There are also non-JVM web frameworks that are non-blocking, for instance NodeJS (JavaScript), Tornado (Python) or CPPSP (C++ Server Pages). This makes the non-blocking approach a current standard in web application development. In studies [16, 17] non-blocking approach was found as the most appropriate for modern web application. This research confirms this statement.

The authors of papers [15, 14] argued that asynchronous, non-blocking invocations always perform better than synchronous, blocking invocations and leads to better scalability. In two conducted experiments, in this research, non-blocking web frameworks always had better performance. Furthermore, from the analysis of RPS for different number of connections (Figures 5.3, 5.5), it can be said that non-blocking web frameworks tend to have higher scalability.

This research corresponds to the study [18], where NIO API was evaluated for High Performance Computing applications. In this research NIO based web frameworks were evaluated in the context of ROA web applications.

Chapter 9

Summary

This chapter summarizes the findings and suggests possible directions for future studies.

9.1 Synthesis

This section revisits each research question and its answer.

RQ1: What are the most popular blocking and non-blocking web frameworks? The most popular three blocking and thee non-blocking web frameworks were selected from the internet ranking and are presented in Table 5.1. The internet ranking showed that non-blocking solutions are the current standard in the web and are the most common approach for new projects.

RQ2.1: What is the maximal performance improvement of using non-blocking instead of blocking calls in ROA based web application? The first experiment showed that maximal number of requests per second that a single server can handle using non-blocking calls is up to 2.5 times higher than when using blocking calls in ROA based web application.

RQ2.2: What is the performance improvement of using non-blocking instead of blocking calls in a chosen existing ROA based web application? The migration conducted in the second experiment resulted in 27% performance improvement for existing ROA based web application.

RQ3: What guidelines for assessing profits of the migration from blocking to non-blocking JVM web frameworks in ROA based web applications can be drawn from the experiments? Chapter 7 contains the prepared guidelines together with the example analysis.

9.2 Conclusions

The problem of performance and scalability of web applications is challenged by most of the software companies. A web application can handle incoming requests using blocking or non-blocking I/O operations. The way that a single server handles requests affects the performance and its ability to scale and depends on a web framework that was used to build a web application. It is especially important for ROA based web application which consist of distributed REST web services.

This research evaluated the influence of web framework's type on the performance of ROA based web applications and provided guidelines for assessing profits of the migration from blocking to non-blocking JVM web frameworks in ROA based web application.

The performance improvement was evaluated on bare and existing web applications by conducting two experiments. The first experiment showed that JVM non-blocking web frameworks provide performance up to 2.5 times higher than blocking web frameworks in bare ROA web applications. The outcome of the second experiment was 27% performance improvement after the migration.

The guidelines were prepared based on the previous experiment results and consultations with the software architects from the company A. The guidelines are fast method for assessing the profits of the migration from blocking to non-blocking web frameworks. The guidelines can be used by product owners and software architects who are working with ROA based web applications which were implemented using JVM web frameworks.

The research successfully convinced the company A to perform the migration on the production environment.

9.3 Future Work

Due to lack of resources the experiments were conducted on private laptops and Amazon AWS instances. There is a need to repeat the experiments on dedicated physical servers to make the results more precise and adequate for the industry. All the prepared bare web applications, benchmark tool and scripts can be found online at Github[1] and can be used to conduct the experiment on other environments.

Furthermore, this study can be with a survey which evaluates what web frameworks are currently used in the industry. Then, the companies that use blocking web frameworks can be asked to apply the guidelines and provide feedback on them.

Those suggestions can improve the guidelines and increase contribution and usefulness of the whole research.

[1] https://github.com/mbilski/benchmark

References

[1] A. F. B. Veal, "Performance scalability of a multi-core web server," in *ANCS'07, Orlando, Florida, USA, ACM*, 2007.

[2] S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010.

[3] M. A. N. C. R. Hashemian, D. Krishnamurthy, "Improving the scalability of a multi-core web server," in *In Proc. of IEEE Global Telecommunications Conference (GLOBECOM), Sym- posium on Advances for Networks and Internet*, 2013.

[4] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton, "Comparing the performance of web server architectures," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, (New York, NY, USA), p. 231–243, ACM, 2007.

[5] D. Bonetta, D. Ansaloni, A. Peternier, C. Pautasso, and W. Binder, "Node.Scala: implicit parallel programming for high-performance web services," in *Euro-Par 2012 Parallel Processing* (C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, eds.), no. 7484 in Lecture Notes in Computer Science, pp. 626–637, Springer Berlin Heidelberg, Jan. 2012.

[6] V. Beltran, D. Carrera, J. Torres, and E. Ayguade, "Evaluating the scalability of java event-driven web servers," pp. 134–142 vol.1, IEEE, 2004.

[7] A. Leonard, "Pro java 7 nio.2," in *Apress*, 2011.

[8] J. B. Shreyas Cholia, David Skinner, "Newt: A restful service for building high performance computing web applications," 2011.

[9] J. J. P. C. R. Joel L. Fernandes, Ivo C. Lopes and S. Ullah, "Performance evaluation of restful web services and amqp protocol," 2013.

[10] M. M. R. Lucchi, "Resource oriented architecture and rest," in *European Commission, Joint Research Centre, Institute for Environment and Sustainability*, 2008.

[11] A. Iyengar and D. Rosu, "Architecting web sites for high performance," *Scientific Programming*, vol. 10, pp. 75–89, Jan. 2002.

[12] U. Praphamontripong, S. Gokhale, A. Gokhale, and J. Gray, "Performance analysis of an asynchronous web server," in *Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International*, vol. 2, pp. 22–28, 2006.

[13] A. G. S. Gokhale and J. Gray, "Response time analysis of an event demultiplexing pattern in middle- ware for network services," in *ICPE'13, ACM*, 2013.

[14] G. Tretola and E. Zimeo, "Extending web services semantics to support asynchronous invocations and continuation," in *IEEE International Conference on Web Services, 2007. ICWS 2007*, pp. 208–215, 2007.

[15] D. L. K. M. Jacques M. Bahi, Raphael Couturier, "Java and asynchronous iterative applications: large scale experiments," 2007.

[16] E. S. P. K. Nabil Benothman, Jean-Frederic Clere and R. Maucherat, "Network performance of the jboss application server," 2013.

[17] S. Drobi, "Play2: A new era of web application development," 2012.

[18] A. S. S. L. Ammar Ahmad Awan, Muhammad Sohaib Ayub, "Towards efficient support for parallel i/o in java hpc," 2012.

[19] H. X. J. N. A. L. B. Upadhyaya, Y. Zou, "Migration of soap-based services to restful services," in *IEEE*, 2011.

[20] M. H. M. C. O. B. R. A. W. Claes Wohlin, Per Runeson, "Experimentation in software engineering," 2012.

Appendix A

Experiments results

Table A.1: RPS on i7. First experiment

Connections	CXF	Jersey	Resteasy	Grizzly	Jetty	Spray
50000	6836	10367	10776	27195	25526	23209
40000	6985	10555	12779	30779	29504	25035
30000	6073	12963	16736	32422	35446	28350
20000	7276	13237	16950	33301	36979	28640
10000	7470	13266	17338	33964	38436	31551
1000	7615	13654	17937	35462	37945	33326

Table A.2: Latency on i7. First experiment

Connections	CXF	Jersey	Resteasy	Grizzly	Jetty	Spray
50000	10931	6848.41	8218.12	3385.84	5208	5822.37
40000	8891	4881.19	4758.01	1674.85	1684	4201.18
30000	5107	2043.478	1588.17	651.69	854	1633.78
20000	2962	1145.56	1241.69	345.28	559	1079.49
10000	1897	652.71	539.31	208.17	278	507.76
1000	244	131.24	177.31	79.51	63	46.67

Table A.3: RPS on AWS. First experiment

Connections	CXF	Jersey	Resteasy	Grizzly	Jetty	Spray
1000	3479	5684	7653	14057	15893	11057
10000	3464	5624	7610	13708	15347	10777
20000	3425	5595	7582	13538	14847	10462

Table A.4: Latency on AWS. First experiment

Connections	CXF	Jersey	Resteasy	Grizzly	Jetty	Spray
1000	553.16	376.43	257.00	98.21	82.19	93.70
10000	4255.50	5323.20	3927.06	690.32	746.01	955.88
20000	18639.46	11381.21	9551.89	1252.63	1474.31	1890.26

Table A.5: RPS on i7 and AWS. Second experiment

API	Blocking		Non-blocking	
	i7	AWS	i7	AWS
1	10107	3.088	12.229	4.818
2	6890	3209	8062	4021
3	5911	3263	7556	3968
4	6750	2345	7609	3355

Appendix B

Hypothesis testing methods

Table B.1: T-test

Name	Description
Input	Two independent samples: $x_1, x_2, ..., x_n$ and $y_1, y_2, ..., y_n$
Degrees of freedom	$n + m - 2$
$\bar{x}, \bar{y}$	Mean values of x_i and y_i samples
S_x^2, S_y^2	Individual sample variances
S_p	$\sqrt{\frac{(n-1)S_x^2 + (m-1)S_y^2}{n+m-2}}$
t_0	$\frac{\bar{x} - \bar{y}}{S_p\sqrt{\frac{1}{n} + \frac{1}{m}}}$

Table B.2: Paired t-test

Name	Description
Input	Paired samples: $(x_1, y_1), (x_2, y_2)...(x_n, y_n)$
Degrees of freedom	$n - 1$
d_i	$x_i - y_i$
S_d	$\sqrt{\frac{\sum_{i=1}^{n}(d_i - \bar{d})^2}{n-1}}$
t_0	$\frac{\bar{d}\sqrt{n}}{S_d}$

Appendix C

Server configuration

Table C.1: Server configuration

Attribute	Value	Description
ulimit		
n	65000	file descriptors
sysctl.net.ipv4		
ip_local_port_range	1024 65535	local port range that is used by TCP and UDP traffic
tcp_tw_recycle	1	allows reusing sockets in TIME_WAIT state for new connections
tcp_max_syn_backlog	8192	the number of outstanding syn requests allowed
java		
server		JVM optimization for servers
Xms	1024m	minimum memory allocation pool for JVM
Xmx	2048m	maximum memory allocation pool for JVM
XX:+UseConcMarkSweepGC		concurrent mark-sweep garbage collection
XX:+AggressiveOpts		performance compiler optimizations