MASTER'S THESIS | LUND UNIVERSITY 2017

Reactive programming and its effect on performance and the development process

Gustav Hochbergs

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2017-33

Reactive programming and its effect on performance and the development process

Gustav Hochbergs
jur12gho@student.lu.se

November 5, 2017

Master's thesis work carried out at Playtech BGT Sports.

Supervisors: Johan Frick, johan.frick@playtech.com
Patrik Persson, patrik.persson@cs.lth.se

Examiner: Jörn Janneck, jorn.janneck@cs.lth.se

Abstract

The focus of this master's thesis is to evaluate the effect of reactive programming on Playtech BGT Sports content server. The effect is evaluated mainly from a performance aspect, but a development process aspect is also taken into consideration. The content server is working in real-time and is required to have low latency and high throughput of processing of data. These characteristics are important to supply customers with the latest information.

A comprehensive theoretical research was conducted in order to be able to implement reactive prototypes correctly and to avoid common pitfalls. Three types of reactive prototypes of the content server were implemented with different execution contexts. The prototypes were tested and the results were compared to the results of a fully synchronous solution. The results showed that reactive programming can increase the performance during high loads. The solutions performed similarly during low load. During high load one prototype stood out with 100% of throughput and low latency. This prototype had an execution context in the thread which subscribed to the result from the executed callback methods.

Reactive programming did in this case increase performance during high loads, but it is worth noting that the execution context is important for the performance. The development process did not change significantly, but reactive programming added complexity to the code and the need for a developer with extensive knowledge in reactive programming.

Keywords: Reactive programming, declarative programming, asynchronous programming, non-blocking execution

Acknowledgements

I would like thank Johan Frick at Playtech BGT Sports for his help throughout the master's thesis with valuable inputs and discussions.

I would also like to thank Patrik Persson at LTH for valuable inputs and discussions throughout the whole master's thesis.

Furthermore, I would like to thank Playtech BGT Sports for making this master's thesis possible.

Contents

Chapter 1
Introduction

Online sports betting has become an immense industry in the 21st century. The selection of sports and number of events that can be bet on is growing rapidly. Having the latest odds is essential for the customer in order to be able to decide whether it is worth betting or not. This is the reason why the time from the moment an odd is updated until it reaches the customers is critical. The expectations for fast odd-updates put great pressure on the underlying system, which should receive, process, persist and distribute odd-updates. Playtech BGT Sports provides systems targeting online betting.

The architecture of the systems which Playtech BGT Sports is providing consists of a client side and a server side. Operations executed on the client-side is done by the web browser by executing JavaScript code. The server-side handles mapping, persisting and distribution of events. The server-side is divided into three different domains: content server, data server and frontend server. The frontend server handles transportation of data to clients, by working with diffs and sending as little data as possible. The data server handles persisting of data and data publish/subscribe. The content server maps data to suitable data models for components residing on the client-side. The data mapped by the content server can be events or updates for events. These are provided by a content API residing at a content provider which can be a betting company for example. For a graphical overview of the architecture see Figure 1.1.

1.1 The Problem

The number of customers and the number of events that can be bet on is rapidly increasing, but everybody expects high performing applications. The correctness of odds displayed is critical for the customers thereby it is also critical for the businesses providing online betting platforms. Thus, the time between the point of receiving of an event or update from the content provider and the point when the user sees the update should be minimal.

To be able to achieve low latency between receiving and distributing the data it is

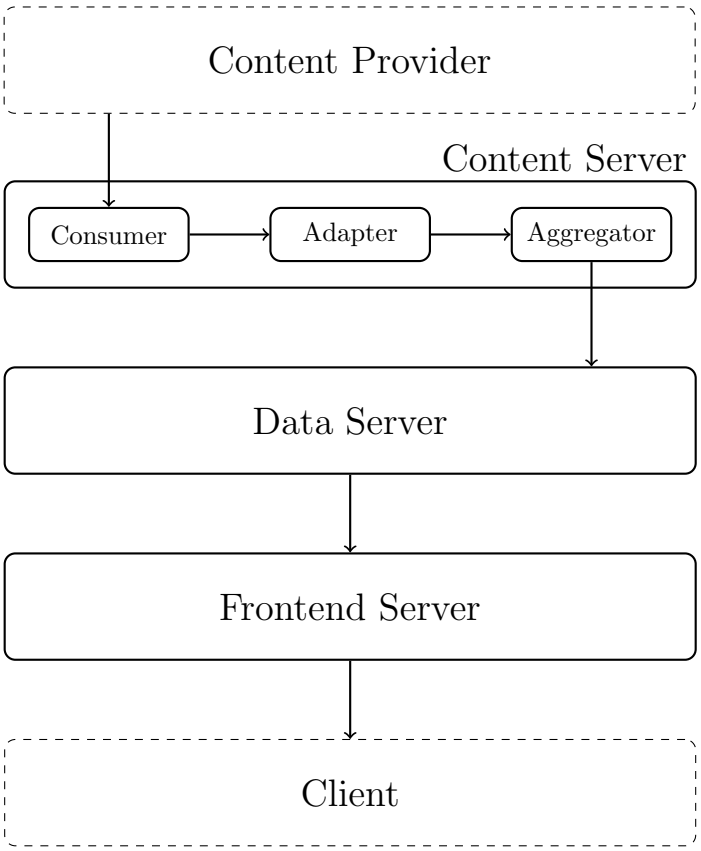

Figure 1.1: Flowchart about the flow of data in the application, from the point where the data is received from the content provider until the point where it reaches the client.

important for each part of the application not to act as a bottleneck. A key part in the application is the content server. The server can be divided into three parts: consumer, adapter and aggregator, see Figure 1.1. After receiving raw data from the content provider, the consumer handles the receiving of the data and the adapter maps the raw data into suitable data models. The aggregator aggregates the data models to lists of data models and more complex data models which are published to the data server. Communication between the three parts of the content server is partially done by synchronous method invocation.

With the growing amount of incoming data finding new ways to improve the application is crucial. One possible solution would be to improve the specs for the machines which the servers are running on, but this solution would not be sustainable in the future because of limited machine power. Playtech BGT Sports is therefore requesting assistance with investigating another possible solution, more specifically the use of reactive programming and asynchronous data streams. This master's thesis is intended give Playtech BGT Sports knowledge to evaluate if the use of reactive programming is a viable solution for the problem.

1.2 Reactive Programming

In the latest years the reactive programming paradigm has received increased attention since more and more applications has become event-driven[10]. The paradigm revolves around propagation of change for continuously time-varying variables. In other words the paradigm revolves about asynchronous non-blocking data stream processing. Reactive programming takes a declarative approach letting developers specify what to do but it is leaving the time of execution to the language. Reactive programming will be covered more in depth in Chapter 3.

1.3 Research Questions

In order to evaluate if the use of reactive programming is a viable solution the master's thesis will address two research questions. The solutions will be assessed from three different views and not only from a pure performance point of view. The research questions are the following:

- How does the use of reactive programming affect the performance in terms of latency, throughput, CPU usage and memory usage?
- How does the use of reactive programming affect the development process?

1.3.1 Scientific Contribution

By answering these questions the master's thesis will hopefully contribute with knowledge about performance of an application with reactive programming applied to the internal logic, but also about the impact on the development process when changing from an imperative style of programming to a declarative style.

1.4 Related Work

Harel and Pnueli defined the term ”reactive system” in 1985 in their paper ”On the development of reactive systems”[17]. In the paper it is determined that two types of systems exist: transformational and reactive systems. Both systems process input and produce output. The difference made between the systems are that a transformational system reacts to inputs from time to time while a reactive system reacts to inputs continuously. The authors' paper defines the term ”reactive” which definition will be used as a background to what reactive is in this master's thesis.

Elliott and Hudak defined the term functional reactive programming in 1997 in their paper ”Functional reactive animation”[14]. The paper proposes a collection of data types and functions to create applications with rich interactive multimedia animations. The proposition was created to make development of multimedia applications easier for developers.

One major criticism against the first proposal of FRP was the use of the pull-based model[14]. In this model a value is needed to be pulled from its source. The criticisms is based on the fact that all computations are needed to be done before the reaction takes place[10]. Complications like time-leaks can occur, meaning that values not yet needed are calculated which is leading to the need of catching up with computations later. Another complication is space-leaks, which occur after a time-leak if space is taken up. These problems mainly arise in lazy typed languages such as Fran. In newer implementations of Fran, such as New Functional Reactive Animation (NewFran), these problems are solved[15].

FRP was precisely defined in 1997. However, many languages and libraries claiming to implement FRP today do not follow the original definition. Languages like Elm and reactive libraries as Reactor are sometime wrongly described as FRP according to Conan Elliott[13]. Most recent implementations lack continuous time and precise and simple denotation. Hence, these should not be considered as FRP according to Elliott.

FRP is an important part of reactive programming and a considerable amount of research has been conducted in this paradigm. However, this master's thesis does not need to discuss FRP further and will leave it as related work.

Data stream processing has wide applications and is not only restricted to reactive programming and FRP. In this master's thesis the source of the stream is coming from inside the application, but this is not a requirement in data stream processing. Streams of data can also come from web services, hardware, user input etc. The common denominator is the processing of streams. The main focus of the master's thesis is data stream processing within an application and therefore a more general approach to data stream processing is not further discussed, but a more exhaustive differentiation between reactive systems and reactive programming is made in Chapter 3. This section focused mainly on FRP because it has a close relation to reactive programming and simply because there is still on ongoing discussion of what FRP is.

Kounev et al. [20] show in their paper "Improving Data Access of J2EE Applications by Exploiting Asynchronous Messaging and Caching Services" how asynchronous processing can be used to improve performance, scalability and reliability. They have identified the bottleneck in e-business systems often is the link between the application server and the database server. The synchronous solution keeps resources locked during database transactions. The performance could be increased by making the transactions asynchronous which means continuing the execution until the response is received instead of waiting for the response. At lower transaction injection rates both solutions handle approximately the same number of transactions per minute, but at higher transaction rates the asynchronous solution handles noticeably higher number of transactions. The usage of CPU by the application server was lower for the asynchronous solution. Thus, reaching hundred percent CPU usage only at higher transaction injection rates. The CPU usage by the database server was approximately equal for both solutions and never reached hundred percent. Since asynchronous processing is a key part of reactive programming there is a potential for reactive programming to increase performance in the case studied in this master's thesis.

Syme et al. [24] describes the asynchronous programming model for F# in the paper "The F# Asynchronous Programming Model". In this paper it is concluded that applications reacting to events are becoming more and more important. Nowadays applications need to handle user interactions, communication with web services, results from paral-

lel computations and cloud computations. The paper also identifies two difficulties with asynchronous programming. Firstly, the OS threads are proved to be costly in terms of resources. Secondly, the difficulty of using callbacks creating ”callback hell”. These two difficulties will be kept in mind during this master's thesis.

Bainomugisha et al. [10] comes to the conclusion in their paper ”A survey on reactive programming” that reactive programming is a programming paradigm suitable for developing event-driven applications. During the latest years reactive programming has received increased attention since more and more applications has become event-driven. The current research on reactive programming has mostly been focusing on functional reactive programming. The authors evaluate languages which implement reactive programming along six axes: the basic abstractions for representing time-varying values, evaluation model, lifting operations, multi-directionality, support for distribution and glitch avoidance. The languages are categorized and compared. They identify glitches to be a problem with distributed reactive programming and concluded that further research is needed in this area.

In recent years reactive programming has also received attention in the Java community with the implementation of reactive libraries as Reactor[5] and the introduction of a reactive standard in Java[2]. In 2015 it was suggested to include reactive streams in future Java JDK release, because a standard for push-based asynchronous programming was missing[2]. It was agreed upon that introducing a standard would make interoperability between libraries easier. Reactive Streams is a reactive abstraction with minimal interfaces to implement a reactive library[6]. The interfaces defined in the Reactive Streams specification is going to be a part of the Java JDK 9 release in the Flow API.

1.5 Outline

In this chapter the background for the master's thesis was presented, as well as a introduction to reactive programming and to related works. The purpose of this chapter was to give a better understanding of the project in general.

In chapter two a method is proposed that will be used to answer the research questions. Chapter three is a theoretical introduction to reactive programming which could help in the understanding of the project's solution. In chapter four the implementation of the prototype and the design choices are presented. Chapter five is about the experimental setup of the performed tests. The results are presented in chapter six and in chapter seven the results and their impact is further explained. Finally in chapter eight the conclusions of this master's thesis project are drawn based on the research questions and the results.

Chapter 2
Method

To find answers for the research questions in Section 1.3, a systematical investigation will be carried out. Not only the understanding of reactive development and implementation is important, but also a method of how to evaluate the development and code must be established. The research questions are of a comparative nature and the method used to find answers is of experimental nature. The two technical solutions to be compared are synchronous programming and reactive programming. The experimental method requires a well-defined method and well-defined metrics. In this chapter the necessary method and metrics to draw conclusions are described.

The exact experimental setup is described in Chapter 5 because some technologies used in the experiments are dependent on the implementation.

2.1 Synchronous Solution

The synchronous solution is implemented according to the existing content server in Playtech BGT Sports. The existing content server is built with an architecture which facilitates scalability and reactiveness. This architecture is removed and the current solution will be fully synchronous with a blocking nature. This solution not only creates useful results for Playtech BGT Sports, but it will possibly also be of more general interest. A comparison between Playtech BGT Sports' solution and reactive programming would not be useful for others without knowledge about Playtech BGT Sports' architecture and implementation.

2.2 Reactive Prototypes

Reactive prototypes are implemented to be able to make a comparison between synchronous programming and reactive programming. To implement prototypes is only possible after conducting a literature survey in reactive programming. The theoretical research in the

subject is also important to avoid common pitfalls and is most definitely essential to implement a truly reactive and non-blocking prototypes. The correctness of the implementation is crucial for the future use of the results.

2.3 Performance Metrics

The experimental method requires quantitative data to be collected for drawing conclusions. To draw reliable conclusions from the results it is important to choose metrics which accurately show how the current application and the prototypes is performing. For example, the disk space usage is not a suitable metric in this case because firstly the application itself is not space consuming and secondly but more importantly nothing is written to disk at this point. By excluding the non-suitable metrics four performance metrics can be identified as applicable for the study. They are CPU usage, memory usage, latency and throughput. These metrics are measured for the synchronous solution and for the reactive prototypes. The results of measurements are then compared, which together gives a complex picture on how the prototypes is performing compared to the current application.

2.3.1 CPU Usage

CPU power is a resource with limited expansion possibilities. If the CPU power supplied cannot process data fast enough the utilization of the CPU needs to be changed to process data faster. Hence, the CPU metric is a good metric not only from an economic view but also from a view of available CPU power.

2.3.2 Memory Usage

Memory, like the CPU, is a resource which has limited expansion possibilities. Therefore, the amount of memory used by the content server is of great interest. The metric gives a good indication if either the synchronous solution or the reactive prototypes have best memory utilization.

2.3.3 Latency

Latency is essentially relevant for the purpose of this master's thesis since odds and match information should reach the users as quickly as possible. From a business perspective low latency provides important information to customers instantly[21] and they are able to make better and quicker decisions about betting. Since the application is working in real-time it is highly important to process the data in the shortest time possible.

Two measurements are of interest in this case. Firstly the measurement of the time period from the moment the raw data reaches the content server until the aggregated more complex data models are published to the data server. Secondly the measurement of the time period from the moment the raw data reaches the content server until the moment the data reaches the users. The first measurement gives an indication of how time-efficiently

the reactive prototype is working and the second measurement indicates where possible bottlenecks are located in the application.

2.3.4 Throughput

The fourth metric is throughput which has an impact on the time until odds and match information reaches the customers. The more data processed during a certain time the better because more information will flow through the content server and reach the customers faster. This metric is not only relevant for the benefits of the business[21] but also for the user experience.

2.4 Performance Tests

Performance tests are the core of this master's thesis since it is building on an experimental method. The tests must be run in an environment where it is possible to measure metrics and get reliable measurements. There are four possibilities to run the tests on: production servers, staging servers, the cloud or a local machine.

Running the performance tests on production server could be argued to be the most suitable solution because the results would reflect performance on a production server where applications are actually running. However, there are two problems with using production servers. Firstly, production servers are always in use and therefore it is difficult to get time to run tests on one. Secondly, production server's hardware can differ therefor no coherent setup exists and this can mean that the same server would need to be available for all the tests. To summarize, It is not only hard to find available production servers for tests but also there is no coherent setup which determines that it is better to run the performance tests elsewhere.

Staging servers can also be suitable to run performance tests on but as with the production servers the availability is a problem. The hardware for staging server is in some cases also weaker than the development computer's hardware. Consistency is needed when running tests and no outside variable, such as running tests on different machines, should affect the results. Therefore, staging servers will not be used for performance testing.

The third option is to run tests in the cloud. Two possible clouds are Google Cloud and Amazon Web Services. Both provide measurements of CPU usage and memory usage so there is no need to use another tool for these metrics. The use of these cloud computing services introduces the complexity of setting up the system for test. The system is dependent on a test content provider, a content server and MongoDB. There are also limits for computing time for both clouds. The master's thesis is under time restraints and therefore running the performance tests on the cloud is not the best option.

Running the performance test on a local computer is the best option in this case. This will give consistency for the tests because the hardware will be the same. There is also no computing limit when using a local computer. The computer which will be used is a development computer and already has the technologies needed to run the content server and MongoDB. The hardware used for running the performance test on is desribed in more detail in the Section 5.1.

2.4.1 Type of Test

Two types of tests are considered in this master's thesis: load test and stress test. Load testing is used to asses how a system performs under a given load where the number of requests is called load[11]. Stress testing is trying to break the system with a big load[11]. How the content server behaves under varying load, which happens naturally on production servers, could also be of interest, but the master's thesis does not consider this aspect because its main interest is on how the synchronous solution and reactive prototypes behaves under different loads.

The master's thesis chooses load testing as a method for performance testing because the content server's performance is compared in different loads and this excludes stress testing.

2.4.2 CPU & Memory Measuring Tools

The tool used to measure CPU usage and memory usage is JVisualVM. The tool is provided by Oracle and is included in every JDK release[3]. The tool provides profiling and also tracing of CPU usage and memory usage. The master's thesis requires measurements of CPU and memory. JVisualVM provides these possibilities and the possibility of tracing the usage over time which makes it possible to export the results for further processing. There are several other profiling tools, but it is not necessary for this master's thesis to deeply analyze thread use nor memory leaks and therefore JVisualVM is considered to be a sufficient tool.

2.4.3 Data Models

The data model used for testing needs to be decided to be able to implement the content server. It does not only affect the implementation but also the processing time of the data. The processing time is longer if the data takes longer time to parse and publish to the data server. After the examination data models of suitable size can be chosen to provide a production like processing time.

2.5 Evaluation of Development Process

Evaluation of the development process can partially be based on quantitative data. The quantitative data measured will be the number of lines of code written. The number of lines of code for the implementation can affect the development process both positively and negatively. By writing less lines of code the development can be faster but it can also add complexity and thereby slowing down development. Therefore not only the lines of code but also the qualitative aspects of complexity is assessed. Added complexity requires more knowledge from developers and is definitely an influencing factor in the decision of using reactive programming.

Chapter 3
Concepts of Reactive Programming

In this chapter the theoretical background of reactive programming is presented to help deeper understanding of reactive programming and the implementation of the prototypes. First the term ”reactive systems” is introduced to give a background to the definition of the term ”reactive” and this is followed by the description of what reactive systems are today. Then reactive programming is thoroughly described and the chapter ends with a short comparison between reactive programming and reactive systems.

3.1 Reactive Systems

In 1985 two categories of systems were proposed by Harel and Puneli[17]. They made a distinction between transformational and reactive systems. A transformational system is defined as a system which responds to input by transforming it and then producing outputs. The nature is that these systems process input and produce output from time to time. However, reactive systems are defined as systems which continuously respond to inputs. Harel and Puneli also points out that reactive systems are present everywhere from cars to phones. The reactiveness can be included by software or chips for example. Both the transformational and reactive systems can be either synchronous or asynchronous.

Later on, these two categories were complemented with a category for interactive systems by Berry in 1989[12]. Berry defines interactive systems as systems which interact with the environment at their own pace. Hence, the distinction between a reactive system and interactive system is the pace in which the input is handled. Reactive systems interact with the environment at a pace dictated by the environment, while interactive systems interact with the environment at their own pace.

The center of increased attention for reactive programming has been especially focused on combining reactive programming and distribution. Reactive programming is further explained in Section 3.3. Good examples for the combination of reactive programming and distribution are web applications and distributed mobile applications. A known

downside of the combination of reactive programming and distribution is that it can lead to glitches[10]. The avoidance of glitches is a factor to be considered when using reactive programming. The definition of glitches is inconsistency of data during propagation of change which leads to unnecessary re-computations. It is harder to avoid glitches in distributed systems because of the network problems such as latency and network failures. The combination of reactive programming and distribution is often called distributed reactive programming or reactive microservices. The Reactive Manifesto defines the characteristics of a reactive system. The Reactive Manifesto is further explained in Section 3.2 to differentiate reactive systems from reactive programming.

3.2 Reactive Manifesto

In 2014 version 2.0 of the Reactive Manifesto was published. A system is defined as components cooperating to provide services for users. The authors of the manifesto believe that ”a coherent approach to system architecture is needed” and continue to define what reactive systems are. Reactive systems are responsive, resilient, elastic and message driven. The expectations from systems have become considerably higher since a few years ago where long response times and several hours of maintenance with servers offline were considered acceptable. In contrast users nowadays treat fast response times and no downtime of servers as a must. The authors of the manifesto believe in order to keep up with the current expectations a coherent approach is needed. [8]

By choosing a reactive system approach these systems will be easier to develop. The systems will be easily modifiable and resistant to failure because they will be flexible, loosely-coupled and scalable. At the same time the systems will give users fast interactive feedback.

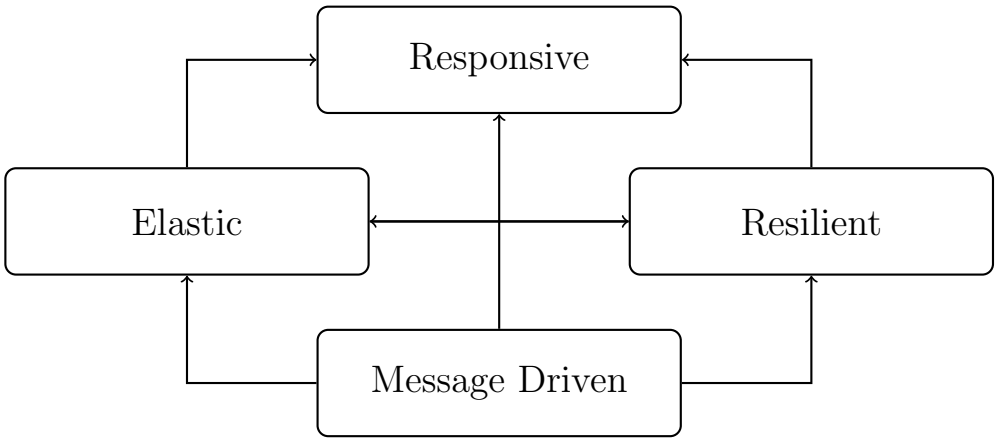

Figure 3.1: Image illustrating how the four characteristics of reactive systems are connected.

The characteristics of a reactive system according to the authors of the manifesto[8] is defined below and Figure 3.1 illustrates how the characteristics are connected.

- **Responsiveness:** For a system to be usable it should provide fast and consistent response times. It makes error detection simple and errors easier to handle. In return this will create a better user experience and thereby the consumers will continue to use the service.
- **Resilience:** For a system to always be available it is important that failure does not affect availability. There are four characteristics for a resilient system. They are replication, containment, isolation and delegation. By replication a component can be executed simultaneously in for example different threads or network nodes. Isolation can be achieved by decoupling. The decoupling can be in time or space. Decoupling in time means that a sender and a receiver communicates asynchronously. Decoupling in space means that the sender and the receiver are running in different nodes in the network. By delegating tasks to other components the original component can oversee the progress and handle failures or perform other tasks during the time.
- **Elasticity:** For a system to always stay responsive elasticity is an important characteristic. This is achieved by increasing or decreasing resources as CPU time and memory in response to change the amount of input. Elasticity facilitates dynamic resource allocation thereby effectively removing bottlenecks.
- **Message Driven:** Asynchronous communication is used by the components to send messages to each other. From this loose decoupling, isolation and location transparency is achieved. Location Transparency is defined as not making any difference between running the system on one or several nodes. In terms of scaling this means no difference is made between scaling vertically and horizontally. Asynchronous message passing helps with delegation of failures and provides the possibility of elasticity and responsiveness by overseeing the message queues and control of back-pressure. If the pace of incoming messages for a component is too high to process it needs to be communicated in a suitable way to the publisher. It is not acceptable for a component to start dropping messages or to stop working. The component should instead communicate this upstream to decrease the pace of messages. This mechanism of handling pressure is called back-pressure.

3.3 Reactive Programming

In the Java community the reactive programming paradigm has achieved more and more attention during the last years. Reactive libraries like Project Reactor and RxJava and the introduction of a reactive standard in the Java 9 API has contributed to the increased interest in reactive programming in Java.

Reactive programming is about data streams and propagation of change. For event-driven application this paradigm is well-suited. A comprehensive research has been carried out on reactive programming. However, mostly focusing on the Functional Reactive Programming paradigm. [10]

Reactive programming resides inside the declarative programming paradigm. In contrast to imperative programming, where the developer uses statements to change a program's state, in declarative programming the developer defines what to do but the time

Listing 3.1: Simple calculation in Java.

```
1 int nbrOne = 7;
2 int nbrTwo = 7;
3 int result = nbrOne * nbrTwo;
```

of execution is handled by the program itself[10]. In Listing 3.1 a simple calculation is shown. Java is an imperative language which means that the code for the calculation would be executed sequentially assigning the value 7 to variable `nbrOne` and then to nbrTwo. The variable `result` will be assigned the value 49. If either of the variables `nbrOne` or `nbrTwo` is changed during the execution of the program the value of the variable `result` will not be updated. If reactive programming would be used the variable `result` would be updated every time either of the variables `nbrOne` or `nbrTwo` is changed.

3.3.1 Evaluation Models

In reactive programming there are two types of evaluation models: pull and push[10]. In most cases which model to use is decided on the language or framework level and the developer has little or no say in the matter.

The pull based model requires the results from computations to be pulled. As the results are pulled it is possible for the puller to decide the pace of which results are received removing the possibility of being overwhelmed with data. The possibility of only pulling the newest value also arises and old values which will never be used can be dropped. One considerable downside of this model is possible latency between the occurrence of an event until the reaction happens. This model is driven by demand and is similar to a iterator pattern.

The push based model instead of demand is driven by the amount of data. As new data is received data can be pushed equally fast making it possible for instant reactions. This is similar to a publish-subscribe pattern. Efficient solutions for resource utilization and to avoid unnecessary re-computations is needed to be able to react instantly. The model fits well for systems where instant reactions are needed and the pace of which data is received is not known, while a pull based model is suitable for systems where data will be available continuously.

3.4 A Reactive Standard in Java

In the last few years extensive work has been concluded in order to provide a reactive standard for Java. The work has resulted in the Reactive Streams initiative which provides a standard for asynchronous stream processing and non-blocking back-pressure[6]. In contrast to the reactive manifesto, where concepts and characteristics of a reactive system were defined, the Reactive Streams initiative's intention is to allow many implementations take part of the benefits of reactive programming.

The initiative defines the most important problem as handling resource consumption when a component is overwhelmed with messages. When a component is overwhelmed

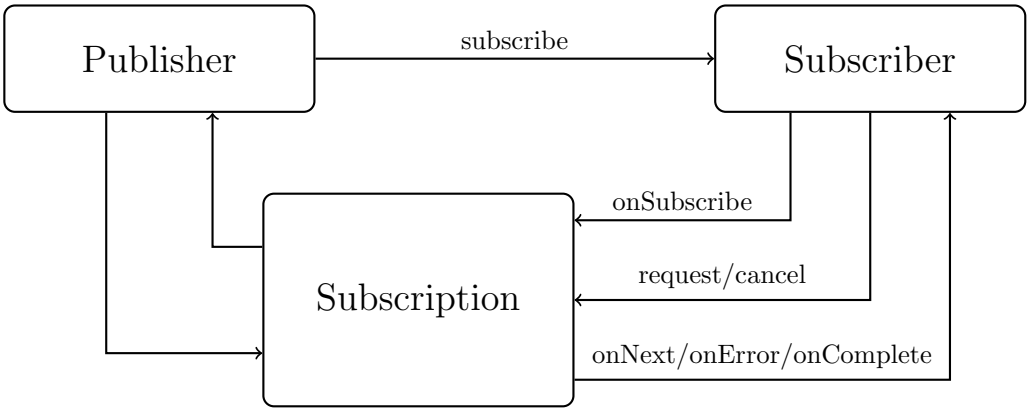

Figure 3.2: Illustration of the flow in an implementation of reactive streams.

message dropping or total failure is not a viable solution. Instead overwhelming pressure should be communicated upstreams by the subscriber to the publisher. The communication of back-pressure should be done asynchronously and not synchronous to benefit from the asynchronous processing. To achieve asynchronous stream processing with non-blocking back-pressure the initiative has set out to find the smallest amount of interfaces, methods and protocols for the implementation.

The Reactive Streams specifications contain four interfaces: `Publisher`, `Processor`, `Subscriber` and `Subscription`. Reactive Streams implement a publish-subscribe pattern. The four interfaces are listed in Listing 3.2 and the flow in an implementation of reactive streams are illustrated in Figure 3.2. Each of the interfaces are described below.

- **Publisher:** A publisher is the provider of data, it can publish an unlimited amount of data to its subscribers. The speed of the publication of data is determined by the subscriber. The interface contains one method called `subscribe` which is used to register a new subscription for a subscriber. The method can be called several times at any point in time. The publisher can stream data to several subscribers.

- **Subscriber:** A subscriber is the receiving end of the data stream from the publisher. The interface has four methods: `onSubscribe`, `onNext`, `onError` and `onComplete`. The method `onSubscribe` is invoked by the publisher when `subscribe` is invoked registering a subscription. No data is streamed to the subscriber until the method `request` is invoked in the subscription. When data is streamed and received by the subscriber the method `onNext` is invoked to process the data. If an error occurs the method `onError` is invoked to handle the error to partially fulfill resilience in the reactive manifesto. At error no further events are streamed until the error is handled. When the stream ends and no further data is streamed the method `onComplete` is invoked.

- **Subscription:** A subscription represents a one-to-one relation between a publisher and a subscriber. The subscription can only be used by one subscriber and when the subscription is canceled it is destroyed allowing resource cleanup. The subscription

Listing 3.2: The four interfaces in Reactive Streams.

```
1  public interface Publisher<T> {
2      public void subscribe(Subscriber<? super T> s);
3  }
4
5  public interface Subscriber<T> {
6      public void onSubscribe(Subscription s);
7      public void onNext(T t);
8      public void onError(Throwable t);
9      public void onComplete();
10 }
11
12 public interface Subscription {
13     public void request(long n);
14     public void cancel();
15 }
16
17 public interface Processor<T, R>
18         extends Subscriber<T>, Publisher<R> {}
```

interface contains two methods: `request` and `cancel`. The publisher does not stream data to the subscriber before the method `request` is called. The method is invoked by the subscriber communicating the number of data to be streamed. The method can be invoked unlimited times. The argument to the method needs to be strictly positive. Using the argument `Long.MAX_VALUE` lets the publisher stream data at any pace. While using an argument as the integer 1 lets the publisher stream one piece of data and lets the subscriber process the data and then the request method can be called again. This design makes it possible for the subscriber to decide the pace of the stream thereby providing back-pressure defined in the reactive manifesto.

- **Processor:** A processor acts as both a publisher and a subscriber. This means that it can both subscribe to a publisher and stream data to a subscriber. The processor is used only between a publisher and a subscriber processing streams of data and streaming the result to a subscriber.

Since the release of Reactive Streams several reactive libraries has started to follow the standard including RxJava, Project Reactor, MongoDB etc.

3.5 Pitfalls of Reactive Programming

Asynchronous programming is defined by non-blocking execution[24]. Reactive programming is about asynchronous processing of data and thereby have to follow the non-blocking execution of asynchronous programming. An asynchronous application is non-blocking in the sense of threads not being blocked during execution to wait for results from other

Listing 3.3: Callbacks easily stacks up creating code which is hard to understand.

```
1 user.login(new Callback() {
2     void onSuccess(new Callback() {
3         void onSuccess(new Callback() {
4             // ...
5         })
6     })
7 });
```

threads. While waiting for a result the thread can continue with its own execution and can handle the result when it is received. This behavior is advantageous when the thread has more to execute. Inversion of control is an important principle for asynchronous programming and also for reactive programming. The principle can be summarized as ”don’t call us, we call you”.

Two problems with asynchronous programming have been the use of OS threads and the difficulty of callbacks[24]. These problems are also important for reactive programming since the execution is asynchronous.

3.5.1 ”Callback Hell”

”Callback hell”, or callback spaghetti, is an expression used to describe the difficulty of using callbacks in larger applications. In asynchronous programming callbacks are used to handle results from asynchronous computations. In large applications with a wast amount of asynchronous computations this can lead to a ”spaghetti” of callbacks making the flow of the application complex[18].

For the programmer the use of callback methods is necessary because it gives the possibility to execute code which is dependent on results from a asynchronous computation. The time when these callback methods are executed is unknown for the programmer which makes it harder to debug. In large applications deeply nested asynchronous computations contribute even more to the the difficulty of not only debugging but also the writing of the code. Computations which are dependent on each other force the programmer to write a lot of boilerplate code[18]. This is resulting in longer development time.

Listing 3.3 shows how easily code becomes hard to understand when callbacks are used. In large applications several callbacks might be used after each other because of depending computations. For example, first the user needs to be authenticated to be able to order. Only after the authentication can information be fetched about the user. After the user information has been fetched can the order be placed because the delivery address is needed from the user information.

3.5.2 Processes and Threads

OS threads are expensive because of allocating system resources and allocating large stacks[24]. Applications implementing asynchronous computations are thread intensive.

Listing 3.4: Example of asynchronous execution where the main thread is blocked.

```
1 ExecutorService executor = Executors.FixedThreadPool(1);
2 Future<Integer> future = executor.submit(() -> {...});
3 Integer result = future.get();
4 /* Do something else */
```

Thus, a programming language with a thread model of one-to-one with OS threads would be CPU and memory intensive. For applications using asynchronous communication a threading model which is not one-to-one with OS threads is important to run smoothly.

When speaking of concurrent programming in Java there are two types of units, processes and threads[4].

Processes provide an execution environment and they are usually synonymous to an application, even though an application can be a collection of processes. Processes are synonymous to OS threads and are expensive to create because each process has its own set of resources, for example memory.

In Java, when talking about concurrent programming, the focus is mostly on threads. Because threads do not take as many resources to create as a process they are sometimes called lightweight processes. Threads cannot exist by their own but instead reside within a process, sharing the process' system resources. Each process has at least one thread.

3.5.3 Non-Blocking

Asynchronous programming is often synonymous with non-blocking[24], in this master's thesis they are also considered to be synonymous. Synchronous means that during execution the application waits for each task to finish before continuing to the next, but during asynchronous execution tasks can be executed on different threads while the main thread continues its execution.

In Listing 3.4 even though the block of code executes a task asynchronously the execution of the block of code is still blocked. The method `Future::get` is blocking and hinders the execution of the current thread from continuation. These cases are needed to be taken into consideration while executing tasks asynchronously. In the example, rather than improving the performance of the application it is decreasing it. This is because the task could be executed in the same thread instead of taking the time and resources to create a new thread. Hence, asynchronous execution is not suitable in all cases.

3.6 Reactive Programming vs. Reactive Systems

The difference between reactive programming and reactive systems is their applications. Reactive programming is applied to the internal logic of components, while reactive systems are applied on an architectural level for systems. Applications implementing reactive programming are highly event-driven which means that events drive the execution forward

instead of a thread-of-execution. Reactive programming facilitates decoupling in time and thereby concurrency. Reactive systems are highly message-driven. The systems facilitate decoupling in space and thereby distribution.

The difference between event-driven and message-driven is defined in the Reactive Manifesto[8]. In the manifesto event-driven is defined as addressable event sources and message-driven is defined as addressable recipients. This means that the subscribers in an event-driven system are attached to the publishers, while in a message-driven system subscribers wait for messages to arrive without the need of attachment to the publisher.

Chapter 4
Implementation

In this chapter the implementation of the reactive prototypes is described. The chapter gives the reader a deeper understanding of design choices which were made and a more detailed description of the problems which were encountered during the implementation process.

4.1 Synchronous Solution

Playtech BGT Sports' current solution is a monolith which has three responsibilities: to consume the content providers API, to map raw data to suitable data models and to aggregate data models to more complex data models. Currently a publish-subscribe system is used between the adapter and the aggregator and between the content server and the data server. For the purpose of this master's thesis and in order to make a comparison between a synchronous and a reactive system, publish-subscribe is removed. This more general approach of the synchronous solution could result in more interesting and more useful findings especially for a more general audience. The change makes the content server fully synchronous and each request will be processed by the same thread.

The current content server is written in Java and makes use of the Spring framework. The content server implemented for this master's thesis is compiled with the Java 8 and uses features from the Stream API and lambda expressions. Features used from the Java Stream API are of synchronous nature and parallel streams are not used.

4.1.1 Push or Pull

The content provider's API can be consumed with either a push solution or a pull solution. Either the API pushes new data to content server or the content server makes pull requests continuously to the content providers API checking for new data.

I made the decision to use a push solution for this master's thesis because of the implementation of both the consumer in the content server and the test content provider becomes simpler. This means that the implementation of the consumer is a REST API which is providing handler methods for processing of requests. The test content provider is an application making requests to the consumer's API. In a push solution it is the test content provider which determines the pace of requests. This makes it easier to control the pace of requests during tests because the content provider performs requests independent of how far the content server has come in processing the requests. In a pull solution the content server pulls data from the content providers API when possible, in other words only when the content server is not overwhelmed.

A push solution is not only advantageous for controlling the pace of data but it also requires less development time since the logic in a pull solution is more complex. The complexity in a pull based solution is mostly in the content provider's API.

4.1.2 Data Models

The impact of the data models is described in Section 2.4.3. Two options exist for processing the data. In the first option the data is first mapped to a simple data model and then published to the data server. In the second option the data is mapped to a simple data model, then published to the data server, then aggregated to a more complex data model and after this the complex data model is published to the data server. I consider the second option as the best choice for this master's thesis because it puts more pressure on the content server and this is necessary to be able to answer the research questions. However, with option two more pressure is put on the content server because the additional composition of the complex data model requires more resource time.

In this case the creation of a complex data model requires two simple data models. Two simple data models are chosen: event and market. The data model event contains information about sport events while the market contains information about odds. The two simple data models are aggregated to a detailed event which contains information about a sport event and several markets for betting on the event.

After examination of production data a conclusion for the number of attributes for each data model was made. The event contains 18 attributes and the market contains 21 attributes. The complex data model's detailed event contains seven attributes from the event and a list of markets with all the market's attributes.

In a production ready content server there are often many more simple data models and these are aggregated to multiple complex data models. For this master's thesis using two simple data models was deemed suitable because the load of data is important for the tests. It does not matter if the load of data is only of two types or several more types since the content server will have to process the same load of data.

The data models impact the implementation of the content server and the implementation of the test content provider. The content server needs two handler methods, one for processing event data and one for processing market data. The test content provider needs to push both event data and market data.

Listing 4.1: The implementation of the controller in the synchronous solution.

```
1 Event event = parse(request);
2 publish(event);
3 DetailedEvent detailedEvent = getDetailedEvent(event);
4 refreshMarkets(detailedEvent);
5 publish(detailedEvent);
```

4.1.3 Processing of Data

The content server consumes the test content provider's API by the content provider making POST requests with raw data in the form of XML. The requests are received by the content server and the raw data is processed inside a handler method. In Listing 4.1 the implementation of the method, which is handling event requests can be seen. The method handling market requests has the same stages as the processing of event requests, but some of the methods in the stages are implemented differently.

Below are each of stages of processing data in Listing 4.1 explained.

1. **Parsing request:** This method parses the request to an event by using the parsing library ngen-xml-parser. The method takes a `HttpServletRequest` as argument and gets the `InputStream` which the parser parses. The method returns an object of type `Event` that contains all the attributes which for example a football match has.

2. **Publish event:** The event is published to the data server which in this case is a MongoDB database. The event needs to be published even if it is used in an aggregator to create a more complex data model. The event can for example be used as a summarization in a list of events. This method only publishes the event and does not manipulate the event and thereby has the return type `void`. Spring framework's `MongoTemplate` is used to save the event.

3. **Get detailed event:** After an event has been published a more complex data model of a detailed event should be composed. This is done by first fetching an existing event from the database and then updating its attributes if one exists, otherwise creating a new detailed event. `MongoTemplate` is used to try to find a detailed event.

4. **Refresh detailed event:** The detailed events' markets need to be refreshed. This is done by finding all markets connected to this detailed event and replacing the existing markets for the detailed event with them. The method has the return type `void` because Java passes the argument by value. This means updating attributes of the detailed event parameter will update the attributes of the detailed event used as argument.

5. **Publish detailed event:** The newly composed detailed event then needs to be published to the database to be available for clients.

The implementation has a one-directional flow of data which means that the handler method does not need to return any data to the content provider.

4.2 Applying Reactive Programming

The master's thesis is focusing on reactive programming. This means that reactive prototypes are implemented to compare performance and development process with the synchronous solution. I see three possibilities of implementing the content server with reactive programming. They are: an implementation using the Java class `CompletableFuture`, an implementation using the Java 9 Flow API or an implementation using a reactive library.

4.2.1 The choices of the application of reactive programming

The purpose of this master's thesis is to investigate the effect of reactive programming. Therefore one reactive implementation is chosen even if comparing the different reactive implementations could also be interesting.

The use of Java 9 Flow API for reactive programming can be discarded. The API provides interfaces rather than complete implementations. Even if a fully implemented publisher is provided the API lacks implementations of the other interfaces. This means that the classes implementing the interfaces are needed to be implemented. However, this can be difficult and time-consuming and it would basically be like implementing a new reative library. The Java 9 Flow API is therefore discarded.

The `CompletableFuture` has many of the features and is a part of the Java API. The class supports chaining and thereby avoids ”callback hell”. Compared to a reactive library the class does not have so many possible execution contexts. By looking at methods provided by the class it is clear that reactive libraries generally have more methods for chaining and processing. More methods are not necessarily better but in this case I would argue that in fact it is especially good because the methods facilitate easier development. The class `CompletableFuture` lacks methods such as map, filter etc. These methods make development easier by not having to implement them separately.

Reactive libraries are chosen for the implementation of reactive prototypes. This is mainly because of their possible customizations and already implemented methods for chaining.

4.3 Reactive Library

There are many reactive libraries for Java such as Akka Streams, RxJava, Reactor etc. It can be hard and almost impossible to determine which reactive library is best, but two reactive libraries are considered to be used for the reactive prototypes in this master's thesis. They are: RxJava and Reactor. These libraries are chosen because Spring 5 has built in support for both libraries.

The two libraries are described in detail below and one of them is chosen for the implementations of the reactive prototypes. The implementation details of the libraries will not be explained as they would prove to be too extensive.

In reactive libraries methods performing operations on data are often called operators so this term will be used in this master's thesis as well.

4.3.1 RxJava

RxJava is part of a group of reactive libraries called Reactive Extensions. The group is all based on the same reactive library first implemented in C#. There are several ports to other programming languages of the library which are part of the group of reactive extensions. The library has been ported to Java, C++, Python etc. RxJava is the name of the implementation in Java. RxJava implements the reactive streams specification since the release of version 2.0 of the library.

The library is influenced by the observer pattern, the iterator pattern and functional programming[7]. In the observer pattern a subject exists with a list of observers which are notified by the subject when a state changes[16]. The iterator pattern makes it possible to iterate over objects in a list without exposing its underlaying structure[16]. The main idea behind using these patterns is to observe a stream of events and to notify when the stream ends. The functional programming provides operators such as map and filter. The library is a fluent API which means that it supports chaining of operators and therefore avoids ”callback hell”. Control over the execution context is provided by the library.

Five main classes are provided by the library. They are: `Flowable`, `Observable`, `Single`, `Completable` and `Maybe`. Two classes are interesting for this master's thesis and they are the classes `Flowable` and `Maybe`. The class `Flowable` emits 0 to N items while the class `Maybe` emits 0 to 1 item. The other classes can be discarded because the content server is usually working with one or more data.

4.3.2 Project Reactor

Reactor is a reactive library implemented for Java even though there are implementations for other languages as JavaScript. The library implements the Reactive Streams specification. The name Reactor comes from the design pattern reactor which the library is influenced by. The reactor pattern simplifies the development of event-driven applications[23]. The library is a fluent API and therefore avoids ”callback hell”. Reactor is influenced by functional programming and has operators such as map, filter etc. Control over the execution context is provided by the library.

The library has two main classes named `Flux` and `Mono`. The class `Flux` has 0 to N elements while the class `Mono` has 0 to 1 element.

4.3.3 Choosing Reactive Library

The libraries RxJava and Reactor are both very similar. Both of the libraries implement the Reactive Streams specification, avoid ”callback hell” by allowing chaining of operators and both support several different execution contexts. The features important for the implementation of reactive prototypes are: avoidance of ”callback hell”, support for different execution contexts and support for back-pressure. Both libraries support these features. I have not found any comparison in performance between the two libraries. It is therefore not possible to make any well-based decision about which library is the most suitable for the implementation of the reactive prototypes. I chose to implement the reactive prototypes with Reactor because it is the default reactive library used by Spring 5.

4.4 Design Choices for the Reactive Prototypes

Java is an imperative programming language and reactive libraries provide a declarative approach. A shift in the approach to the processing of data requires reconsideration of the implementation. This means that the handler methods in the content server need to be rewritten. Below the design of the reactive prototypes is explained.

4.4.1 Identify Sequential Parts

Sequential parts of the synchronous solution are identified before implementing the reactive prototype. Each of the identified parts will be chained with operators making sure they are executed sequentially for all data. The implementation of the synchronous processing is listed in Listing 4.1. There will be five stages of the implementation: parsing request, publish event, get detailed event, refresh markets and publish detailed event. Both the handler method for events and the handler method for markets are consisting of the same parts.

The synchronous parts are independent of each other in the sense of that it is not necessary to know what has been executed before or will be executed after. This fact is important for the reactive implementation where operators process data but does not have any knowledge about what has been processed before or will be processed after. The operators processing data are further explained in Section 4.4.3, but they can generally be described as having one type of data as input and one type of data as output. Each of the synchronous parts has one argument and are therefore good to use with operators to write readable code. Otherwise lambda expressions are necessary to use which can make the code less readable.

4.4.2 Reactor Class

The reactive library Reactor has two main classes: `Flux` and `Mono`. The class `Flux` handles streams of data with 0 to N elements while the class `Mono` handles streams of data with 0 to 1 elements. The classes handle streams with data of possible different sizes and this is reflected in the operators available for the classes. For example, the operator `collectToList` is only available in the class `Flux` because it does not make sense to collect a stream of 0 to 1 elements into a list.

The decision of which class to use is based on the design of the synchronous solution. The solution uses a push model which means that events and markets are pushed to the content server. Markets and events are usually pushed from the content provider instantly when ready. This means that markets and events are pushed one by one because when a push is ready it is not dependent on other pushes. The class `Mono` is therefore suitable to use.

In a pull model data is pulled by the content server from the content provider. This means that the communication of data between the content server and content provider is dependent on the content server and not on the content provider. There can be a delay from

the moment data is ready at the content provider until the data is pulled. If several data is ready during this delay then they will be pulled in a list from the content provider. In this case the class `Flux` would have been a suitable choice because multiple data would need to be handled at once.

I have not found anything pointing to any performance differences between the classes and therefore the class `Mono` is chosen based on its suitability with the push model.

4.4.3 Operators

The class `Mono` has operators which operates on single items.

The class has multiple different operators which should be used in different cases. Operators suitable for the reactive prototypes are identified and used in the implementation. Two operators are identified to be suitable by studying the documentation for the class `Mono`. The operators are `map` and `doOnNext` and they are described below.

- **`Mono::map`:** The operator is triggered when the previous mono emits an element. The operator takes an argument of type `Function`. The interface `Function` is supplied by the Java API and has the type parameters `T` and `R`. The type parameters specify the argument type and the return type of the function. The operator returns a new mono with the type parameter `R`. The method supplied as an argument to the operator has an argument of type `T` and returns an object of type `R`. This means that the method can take one type of object as argument and return a different type.

- **`Mono::doOnNext`:** The operator is triggered when the previous mono emits an item. The operator takes an argument of type `Consumer`. The interface `Consumer` is supplied by the Java API and has the type parameter `T`. The operator returns a mono with the type parameter `T`. The method supplied as an argument to the operator takes an object of type `T` as argument and has the return type `void`. The object which the mono emits is of type `T`.

The operators used were identified by placing the sequential parts of the synchronous solution into categories. The first category of methods takes an object as an argument and returns an object of a different type. The second category of methods takes an object as argument and manipulates the object. The third category of methods takes an object as argument and does not manipulate it in any way. The methods `parse` and `getDetailedEvent` is placed in the first category. The method `refreshMarkets` is placed in the second category. The method `publish` is placed in the third category.

The operator `map` is suitable for use with the first category of methods because operator can handle methods which return an object of different type than the type of the argument. The operator `doOnNext` is suitable for use with both the second and third category of methods. This is because the methods has the return type `void` and the same object emitted from the previous mono should be passed on to the next mono.

Data does not start to flow with just using these operators but requires the operator `subscribe` to be chained before the data starts flowing. Operators can also be used to change the execution context. These types of operators are described in Section 4.4.4 and Section 4.5.

4.4.4 Scheduling

The execution context of the asynchronous tasks can be changed in Reactor. By default the execution is in the same thread as subscribe is invoked. The execution context can be changed to suit different needs. The reactive library Reactor is called concurrency agnostic which means that it is up to the developer to decide which execution context to use. Reactor helps with concurrency by supplying alternative execution context to the default. There are five additional types of scheduling of the execution context to the default. These can be accessed with static methods in the class `Schedulers`. The class gives access to the following contexts: immediate, single, elastic, parallel and timer. The execution context can be the following options:

- **immediate:** The execution is done in the current thread. This is the default execution context.
- **single:** The execution is done in a single thread which is the same for all tasks. The single execution context can also be used to create a dedicated thread for each task by calling the static method `newSingle` in the class `Schedulers`.
- **elastic:** The execution is done in an elastic thread pool which increases or decreases the amount of worker threads after demand. Worker threads are reused when a task has finished its execution.
- **parallel:** The execution is done in a fixed size thread pool. The name parallel refers to the possibility to execute tasks in parallel and therefore creates as many worker threads as there are CPU cores. Note that by choosing this execution context does not make tasks execute in parallel by itself. To have a real parallel execution context the operator `parallel` must be chained before the execution context is changed.
- **timer:** This context can be used to schedule tasks in the future.

Except from the default execution context the execution in an elastic thread pool and in a fixed thread pool is relevant for this master's thesis. Comparing the default execution context with the thread pools is interesting because changing or creating new threads is not always advantageous for performance. The results can give a good indication which execution context is advantageous for these types of tasks. The elastic execution context will resize the thread pool after demand and this can affect the performance negatively. The creation of new worker threads and the obligation to handle more worker threads might consume resources. A fixed thread pool does not have to resize the thread pool but the limited threads might instead block tasks from instantly starting their execution.

Execution context can be provided by two different operators:

- `publishOn:` the operator defines the execution context to the subsequent operators. This operator can be chained defining the execution context for operators until another `publishOn` is chained.
- `subscribeOn:` the operator defines the execution context of the previous operators. This operator cannot be chained and the first subscribe operator determines the execution context.

Listing 4.2: The reactive implementation.

```
1 request.getBody().collectList()
2       .map(this::parse)
3       .doOnNext(this::publish)
4       .map(this::getDetailedEvent)
5       .doOnNext(this::refreshMarkets)
6       .doOnNext(this::publish)
7       .subscribe();
```

The operator `subscribeOn` is used in the reactive prototypes to determine the execution context because multiple different execution context are needed.

4.5 Implementation of the Reactive Prototypes

Spring 5 will be used for the implementation of the reactive prototypes of the content server. At the moment of the implementation Spring 5 is only available in snapshots releases and milestones releases. This is probably not suitable for use in production but can be used for this master's thesis. This because the functionality used in the implementation of reactive prototypes is implemented. Non-blocking I/O is supported in Spring 5 and can be utilized with application servers as Netty and Tomcat. Both of the application servers support servlet 3.1 which introduces non-blocking I/O. The performance of an application is affected of both the framework which is used and the application server[25]. The impact will be discussed in Chapter 7.

To serve the best interests of this master's thesis's purpose Tomcat will be used. The goal is to make a comparison between synchronous programming and reactive programming and not to make a comparison between application servers. Variables in the experiment, which are not intended to be investigated, should be static between the synchronous solution and the reactive prototypes.

Two REST controllers are implemented with Spring. Each of the controllers has handler methods for processing of POST requests. The processing of data should be done asynchronously and therefore the operators discussed `map` and `doOnNext` is used. The implementation of the handler method is different from the synchronous solution, but the implementation of the methods in each controller remains the same since the processing of data is the same. The implementation of the reactive prototype with the default execution context is listed in Listing 4.2.

Three different execution context for the asynchronous tasks will be tested. Changing the execution context from the default is done with the operator `subscribeOn`. In Listing 4.2 the operator `subscribe` will be exchanged for the following two operators:

- `.subscribeOn(Schedulers.elastic())` for the elastic thread pool and
- `.subscribeOn(Schedulers.parallel())` for the fixed thread pool.

The first row of code in Listing 4.2 prepares the request into a a mono. The variable request is an object of type `ServerHttpRequest` and is provided by the Spring API. The class contains information about the HTTP request. The body is extracted from the HTTP request with the method `getBody` which returns a flux with the type parameter `DataBuffer`. All the `DataBuffer`s emitted from the flux are collected into a list with the method `collectList`. The method returns a mono with the type parameter `List<DataBuffer>`. A mono has been extracted from the HTTP request and is now ready to be processed.

The content server has a non-blocking I/O and this is why the body of the request is a `Flux<DataBuffer>`. All data sent with the request might not be contained in one buffer. The parsing requires all of the data and therefore it is a necessity to collect the buffers into a list. The invocation of the method `collectList` is blocking. However, this does not play a significant role for the master's thesis since it is the processing of the data and not the receiving of request which is investigated.

Chapter 5
Experimental Setup

This chapter is about the experimental setup and the execution of the tests. A content provider was implemented to execute the tests, this is also described in detail in this chapter. How the latency and throughput was measured is dependent on the solution and therefor it is discussed in this chapter as well.

5.1 Hardware

The tests are run on a local computer. It is a MacBook Pro running OS X El Capitan. The processor is a 2.2 GHz Intel Core i7 processor and the RAM is 16 GB 1600MHz DDR3. As the processor has hyper-threading there are four physical cores but eight logical cores. The hardware of the computer could in some extent have an impact on the test results. The hardware needs to have enough power to load test the content server. However, the results from the comparison between the synchronous solution and the reactive prototypes is relative. If the solutions perform differently they should do so on other machines too.

5.2 Content Provider

A test content provider is implemented to carry out tests. The content provider will push data in a certain fixed rate. The test content provider is implemented with Spring and uses the class `WebClient` to perform POST requests.

Spring has an annotation called schedule which schedules tasks. The tasks can be scheduled to run at certain intervals independent of the fact that the last task has finished or not. The scheduling can be done in intervals of milliseconds which is perfect for the performance tests carried out in this master's thesis.

Instead of implementing a content provider a tool like JMeter can also be used. JMeter is a tool for load testing servers. The tool has intuitive interface and good load generation

capabilities according to Bhoomit P. et al in the paper ”A Review Paper on Comparison of SQL Performance Analyzer Tools: Apache JMeter and HP LoadRunner”[22]. JMeter was considered to use but was discarded because of the lack of ability to create an even load for performance tests in this master's thesis.

5.3 Measurement of Metrics

The metrics CPU usage and memory usage are measured with the tool JVisualVM. A plug-in called Trace-Monitor Probes is used to log the metrics during the execution. The tool measures the CPU and heap size every second. The CPU is measured in percentage and the heap is measured in bytes.

The latency is fairly difficult to measure with any tool. JVisualVM has profiling tools which can measure execution time for specific methods but this is harder to do for the reactive prototypes because several methods are to be measured. The tasks are also executed asynchronously which makes it more difficult with JVisualVM. The latency and throughput measurements are therefore done by logging time. Several points of logging will be done, specifically when: an event or a market has been received, an event or a market has been published and a detailed event has been published.

The static method `System.currentTimeMillis` will be used to log the time in milliseconds. The resolution of the time returned by the method is dependent on the system[19]. Since Java 5 there is also a method for returning the time in nanoseconds. The method is called `System.nanoTime`. It might be able to measure the time in nanoseconds but it is only as accurate as the underlaying system clock[19]. As mentioned before, greater resolution than milliseconds is not necessary in this master's thesis. Therefore the method `System.currentTimeMillis` will be used.

5.4 Execution of Tests

Two types of tests are run separately. The first for measuring CPU usage and memory usage. The second for measuring the latency and throughput. Each type of test is run multiple times with the content server pushing data in intervals of 200, 100, 75, 50, 25, 10, 5 milliseconds. This means that requests are pushed with the rates of 5, 10, 13.33, 20, 40, 100, 200 requests per second.

The intervals were chosen to cover vast varieties of pressure put on the content server. Around 200 live events with belonging markets are not unusual during high season. Therefore 200 events with 10 markets belonging to each of them will be used. Every other push from the content server will be an event and every other a market. With an interval of 5 ms each detailed event will have two updates a second. How the events and markets are pushed does not play a significant role since the same processing of data is needed.

The test for measuring the CPU usage and memory usage was setup by starting the content server and then starting the tracing of metrics in JVisualVM. Then the content provider was started. The tests were ran for 15 minutes. The tracer in JVisualVM is then stopped and the applications are shutdown.

The test for measuring latency and throughput was setup by first updating the code with logging of time. These tests were started by first starting the content server and then starting the content provider. The applications were shut down after 5 minutes.

Chapter 6
Results

The results of the research conducted in this master's thesis are presented is this chapter. This chapter specifically focuses on the results of the performance tests, on the development process of the implementation of the reactive solution.

For easier understanding I used specific names for the solutions which are the following:

- **Mono:** for the reactive prototype with the execution context in the thread that invokes the operator subscribe.
- **Mono Elastic:** for the reactive prototype with the execution context in an elastic thread pool.
- **Mono Fixed:** for the reactive prototype with the execution context in a fixed thread pool.
- **Synchronous:** for the synchronous solution.

6.1 CPU Usage

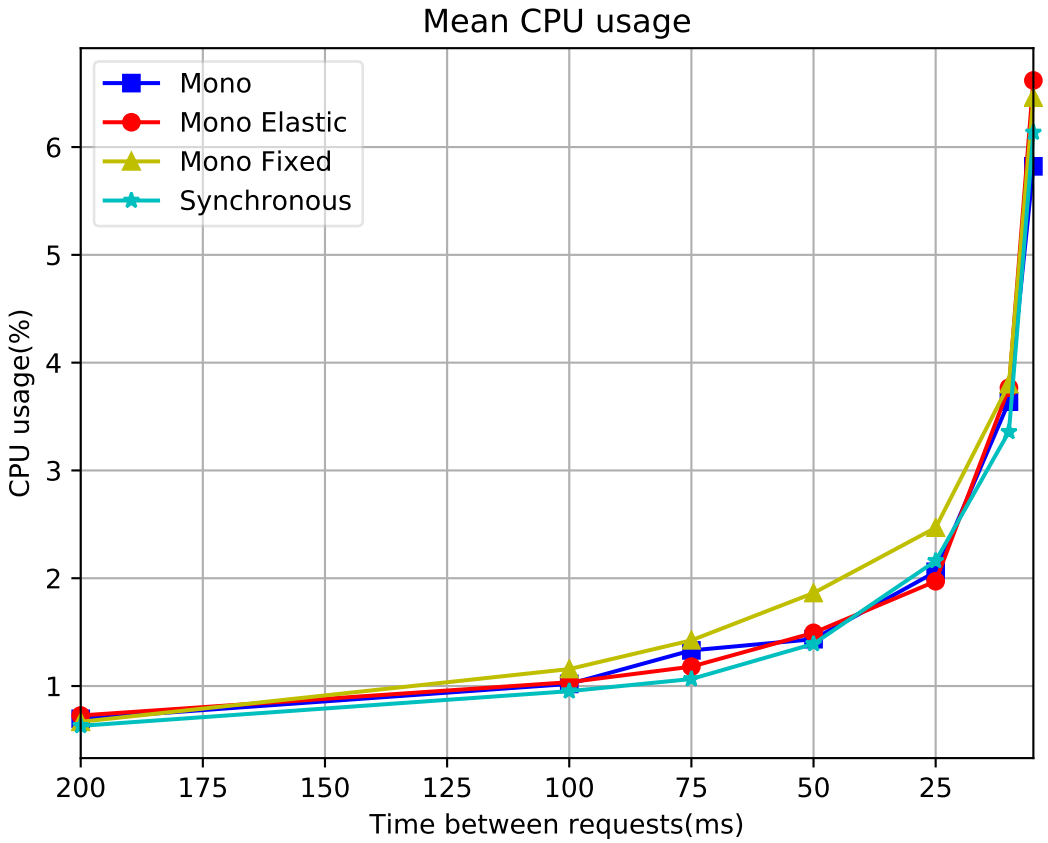

Figure 6.1: Graph displaying the results from measurements of CPU usage. The x-axis is the interval between pushes in milliseconds. The y-axis is percent of the CPU usage.

The mean CPU usage given in percent for each solution is presented in Figure 6.1. The CPU usage is fairly similar for all solutions independent of the interval of pushes. A small difference in the measurements can be noticed when the interval is 5 ms, then the mono prototype has approximately half a percentage less CPU usage than the mono elastic prototype.

6.2 Memory Usage

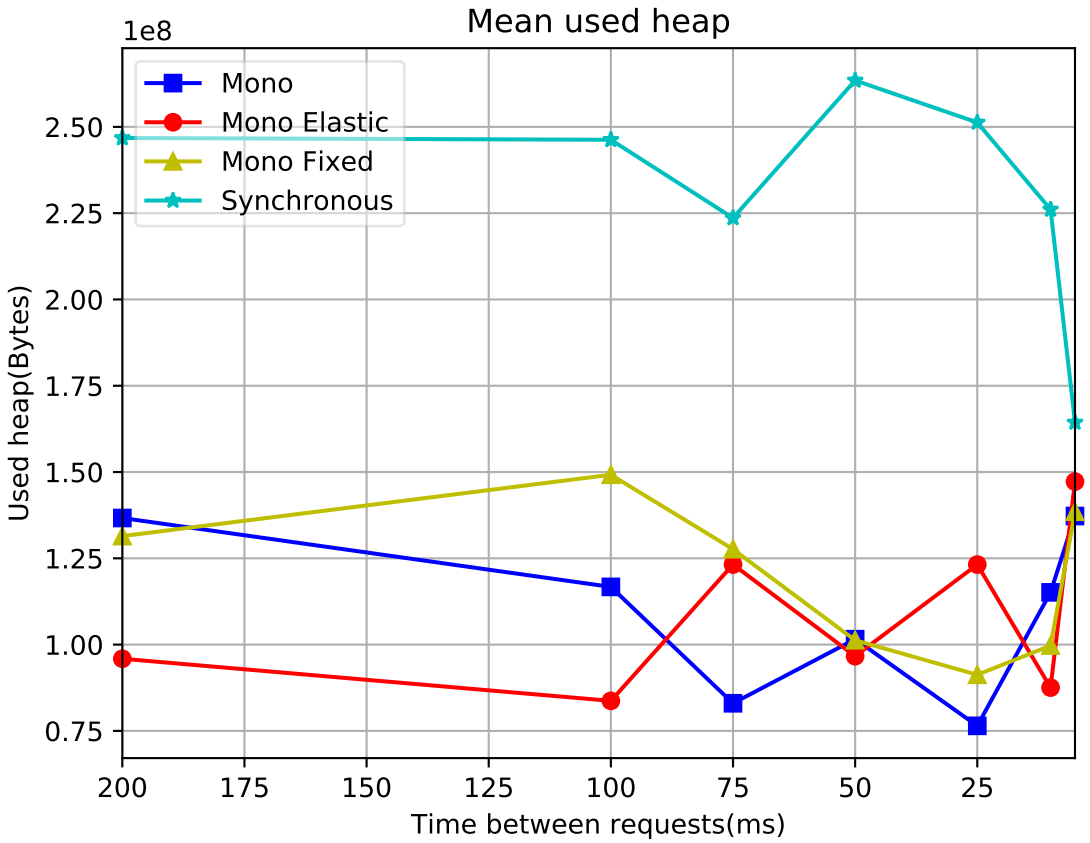

Figure 6.2: Graph displaying the results from measurements of memory usage. The x-axis is the interval between pushes in milliseconds. The y-axis is heap use in 10^8 bytes.

The mean heap usage is presented in Figure 6.2. The reactive prototypes have less heap usage in all intervals than the synchronous solution. The mean heap size varied much from test to test for the same solution.

What cannot be seen in the figure is that when the content provider is started the heap size grows quickly, but decreases and stabilizes after a few minutes. The stabilization is happening much slower for the synchronous solution for all intervals between 200 and 10 milliseconds. The heap size is still decreasing after 15 minutes for the synchronous solution. When the content provider is pushing data every 5 milliseconds the heap size for the synchronous solution decreases and stabilizes quicker. The heap size for the synchronous solution and each prototype has a saw-tooth pattern.

6.3 Event

6.3.1 Latency

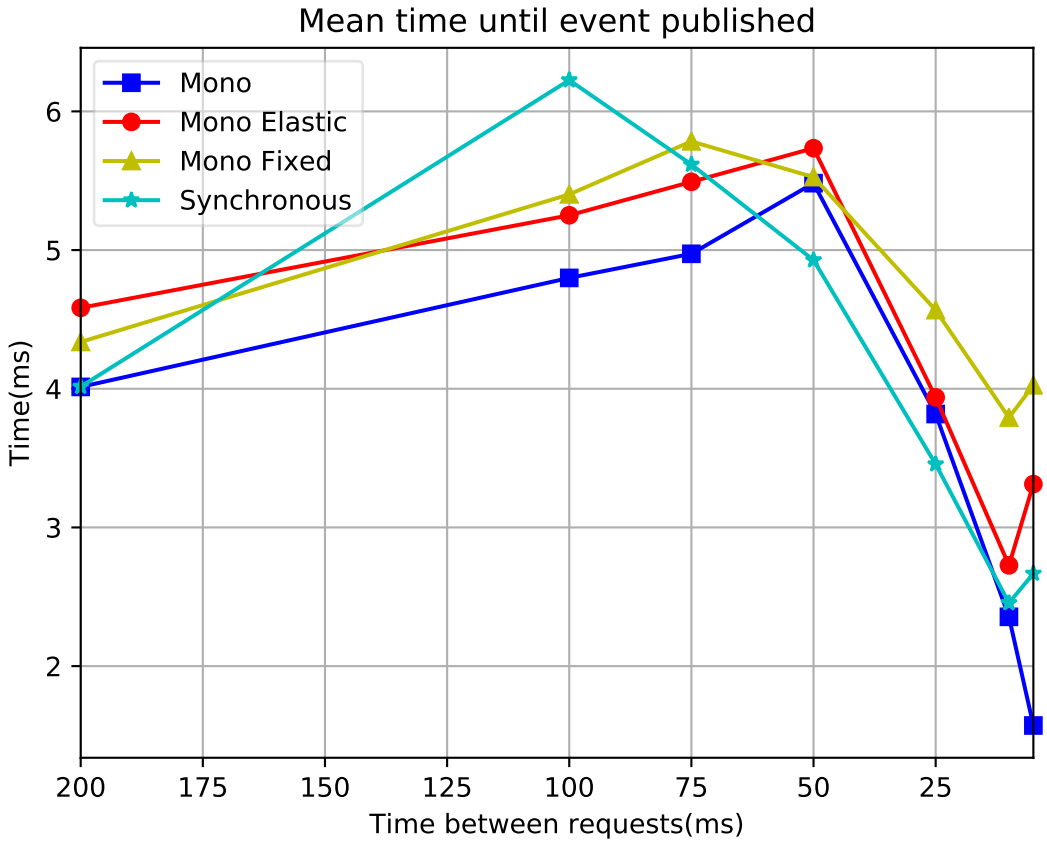

Figure 6.3: Graph presenting the mean latency for each solution until an event is published. The x-axis is the interval between pushes in milliseconds. The y-axis is time in milliseconds.

The mean latency until an event is published is presented in Figure 6.3. The latency for the solutions increases from the interval 200 ms until the interval 50 ms. With less of an interval than 50 ms the latency decreases. Between interval of 10 ms and 5 ms the latency increases for all solutions except for the mono prototype.

6.3.2 Latency of Detailed Event

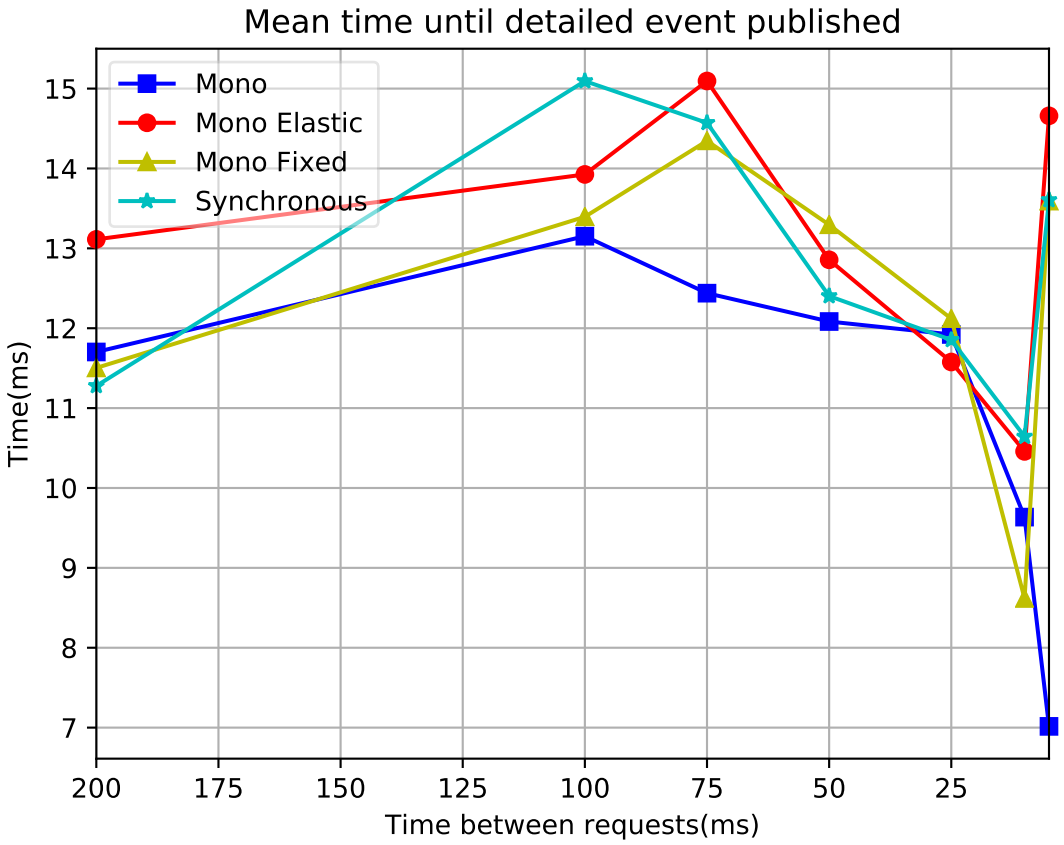

Figure 6.4: Graph presenting the mean latency until a detailed event was published. The x-axis is the interval between pushes in milliseconds. The y-axis is time until detailed event is published in milliseconds.

The mean time until a detailed event is published after processing an event is presented in Figure 6.4. The latency for the different solutions is similar until 5 ms. Between intervals of 10 ms and 5 ms the solutions increase in latency except for the mono prototype which decreases in latency.

6.3.3 Throughput

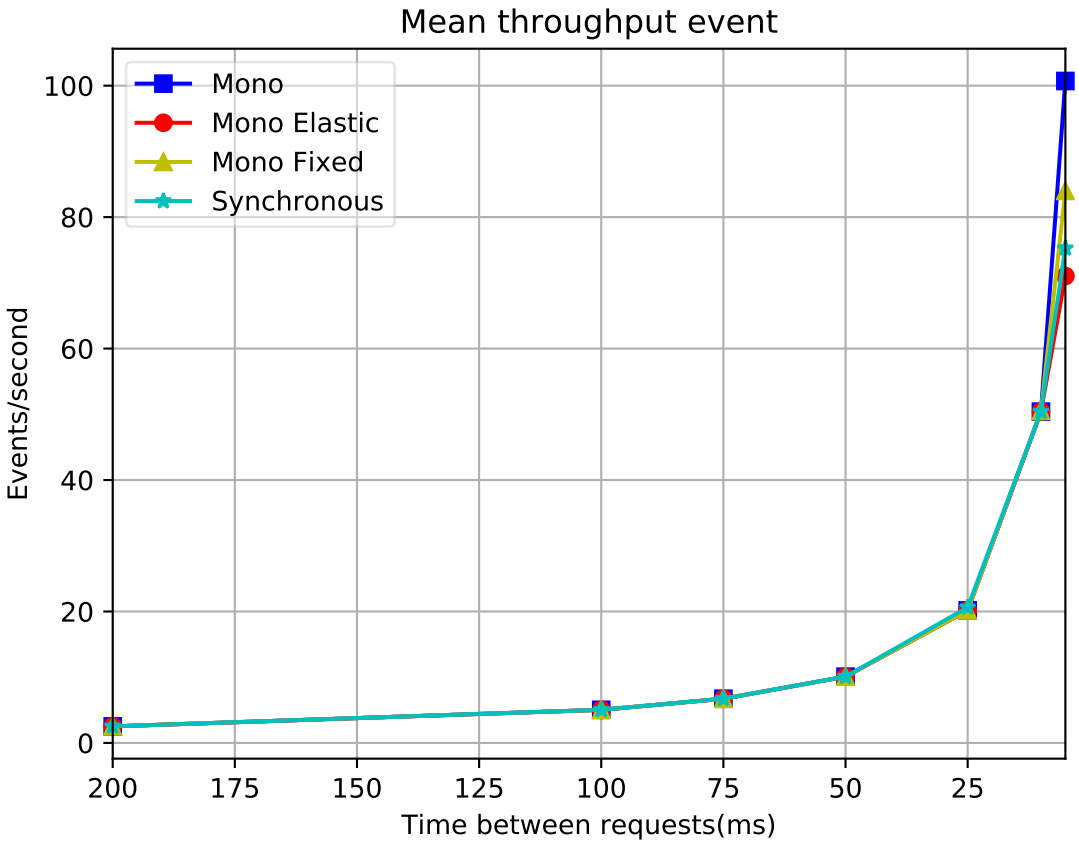

Figure 6.5: Graph presenting the mean throughput for each solution. The x-axis is the interval between pushes in milliseconds. The y-axis is number of events processed per second.

The mean throughput for events for the different solutions are presented in Figure 6.5. All solutions has 100% throughput until intervals of 5 ms. At intervals of 5 ms only the mono prototype has 100% throughput. The other solutions process less events than are pushed to them.

6.4 Market

6.4.1 Latency

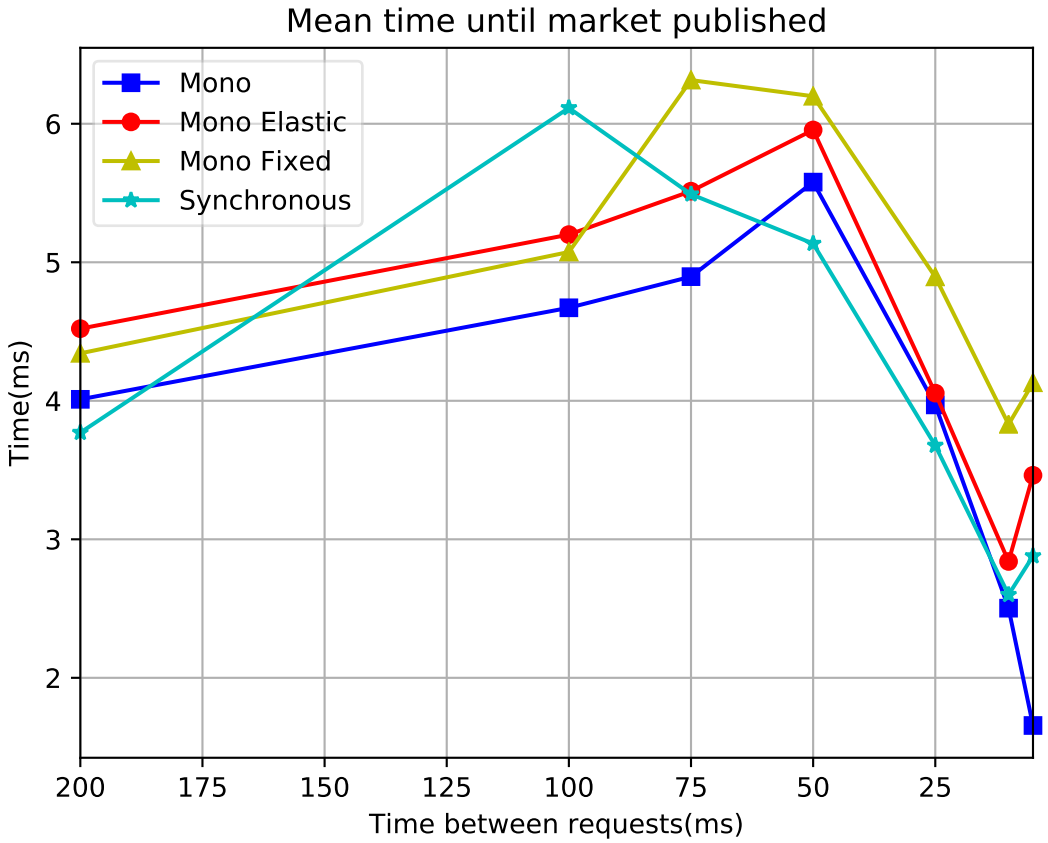

Figure 6.6: Graph presenting the mean latency until a market is published. The x-axis is the interval between pushes in milliseconds. The y-axis is mean latency in milliseconds.

The mean latency for each solution is presented in Figure 6.6. The latency increases slightly after the interval 200 ms until the interval 50 ms. After the interval the latency decreases for all solutions. From interval 10 ms to interval 5ms increases the latency for all solutions except the mono prototype.

6.4.2 Latency of Detailed Event

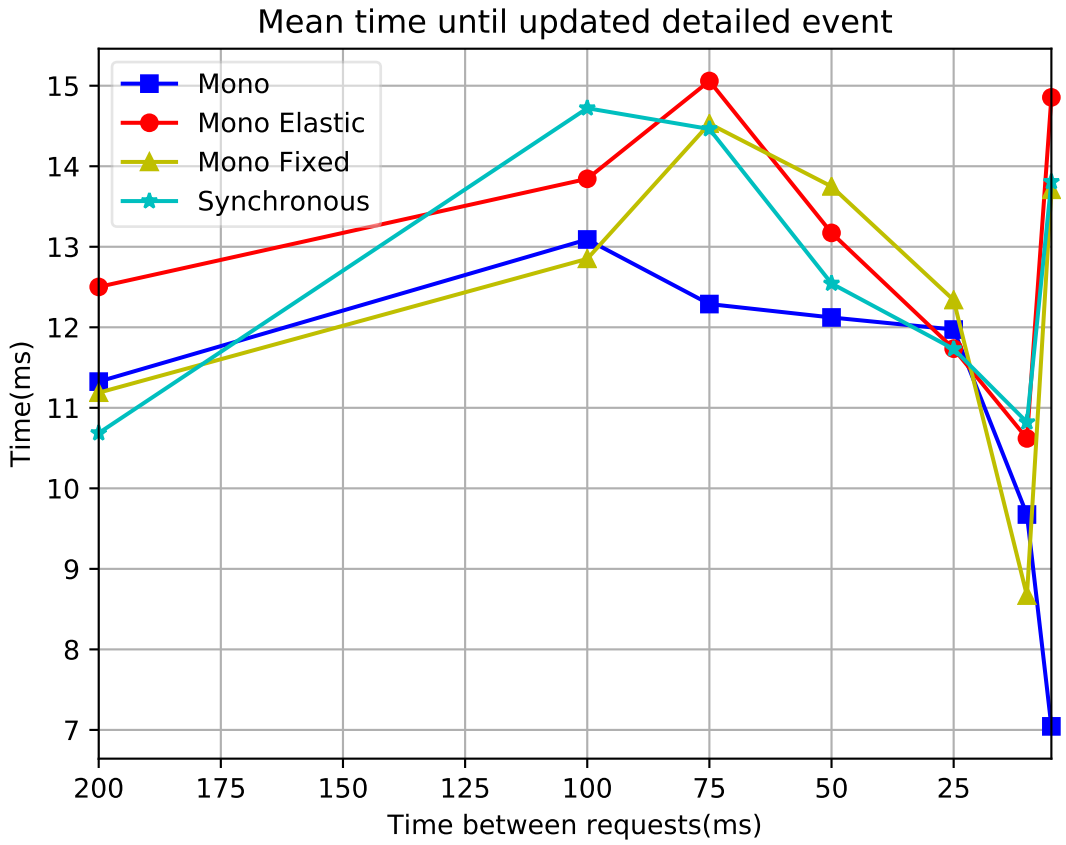

Figure 6.7: Graph presenting the mean latency until a detailed event is published after processing a market. The x-axis is the interval between pushes in milliseconds. The y-axis is time until a detailed event is published.

The latency until a detailed event is published after processing a market is presented in Figure 6.7. The latency for all solutions increases from 200 ms untill 75 ms and thereafter decreases. After intervals of 10 ms the latency increases for all solutions except the mono prototype.

6.4.3 Throughput

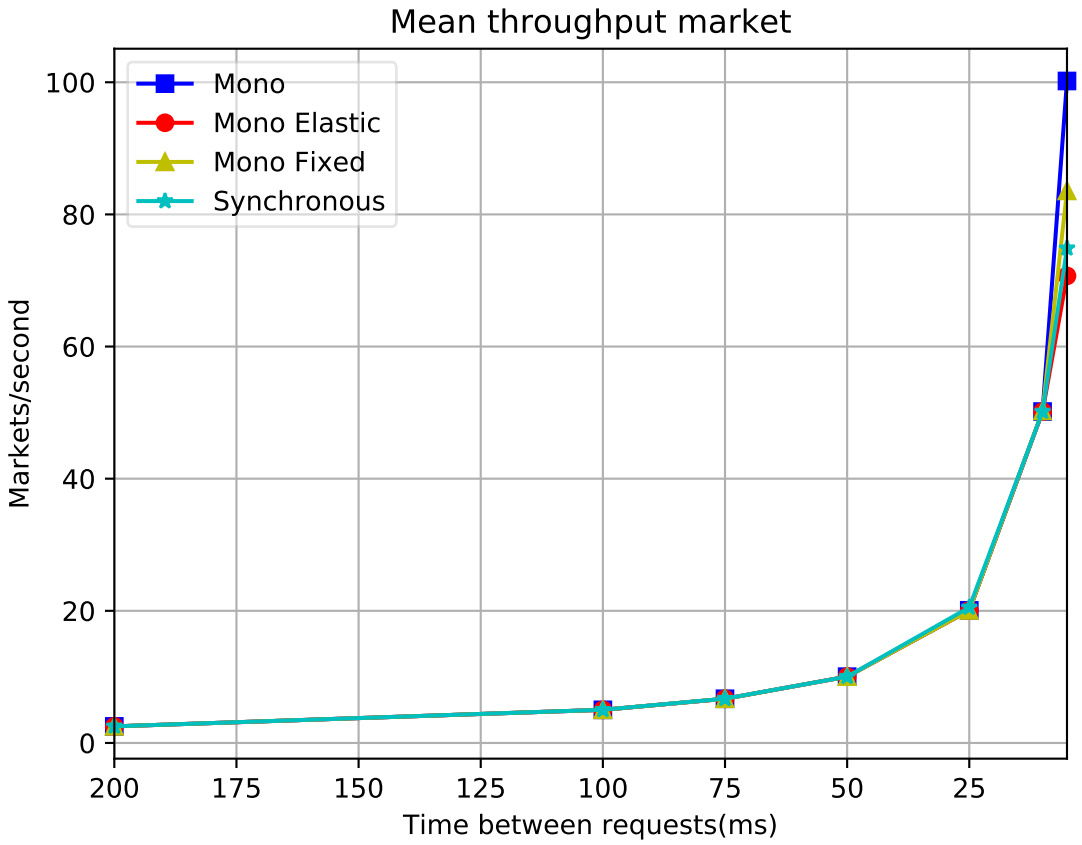

Figure 6.8: Graph presenting the mean throughput for all solutions. The x-axis is the interval between pushes in milliseconds. The y-axis is the number of markets processed per second.

The mean throughput for processing markets is presented in Figure 6.8. All solutions has 100% throughput until intervals of 5 ms. When pushes are made with 5 ms interval only the mono prototype has 100% throughput. The other solutions process less markets than the content server has pushed.

Table 6.1: The table presents the number of lines of code written for each solution.

	Total	Handler method
Synchronous	402	5
Reactive	407	1

6.5 Development

In this case reactive programming did not have a big effect on the development process. Developing the content server using reactive programming did not change the code much from the synchronous solution. The methods used by the controller are still implemented the same way except for the parsing method which has a parameter of type `List<DataBuffer>` instead of `ServletHttpRequest`. By changing the type of the parameter for the parse method the implementation had to change to extract the input stream from the `DataBuffers`. It was not necessary to change the other methods used for processing.

The implementation of the controller method did change but not the ordering of invocation of methods. By changing the execution context from synchronous to asynchronous the invocation of the methods rather became callbacks. No ”callback hell” was experienced instead callback methods were chained with operators.

The implementation of the prototype would have been impossible without a comprehensive knowledge about reactive programming and the reactive library. The acquired knowledge also played a significant role in finding and fixing bugs.

It was more difficult to perform debugging due to asynchronous execution especially in the case of bugs which only occurred when a vast amount of pushes were made. One of the difficulties lies in the fact that the results from previous callback methods are not stored. This means that only variables in the current callback method are visible and accessible. It is possible to put breakpoints in all callback methods to trace the processing of data.

There were also some problems with the first reactive prototypes which used Reactor Netty. Specifically, notifications about errors and warnings for memory leaks were received. Most of the problems arose from Netty methods which I had no control over. After several failed tries to fix the problems that could have been caused by the implementation solutions I made the decision to use Tomcat instead. Netty uses reference counting and, if memory is not released before the garbage collector removes the object, a memory leak occurs. Tomcat does not use reference counting and therefore no problem can arise considering this kind of memory leaks. The memory leak only occurred with reactive programming and I was unable to debug.

The number of lines of code written during the development is presented in Table 6.1. The synchronous solution has approximately the same total number of lines of code written as the reactive prototypes. The reactive solution has less lines written in handler method than the synchronous solution.

Chapter 7
Discussion

This chapter discusses the results presented in the previous chapter. A connection between the results and theory is made and the effect of the design choices is also discussed.

7.1 Performance

The application server Tomcat has a maximum of 200 threads dedicated to request processing and has 10000 of maximum non-blocking connections[1]. In the case of a push model, where data is pushed one-by-one to the content server, a new thread is created for each processing of data. Hence, processing of requests is done asynchronously independent of reactive programming. In this case reactive programming contributes to the fully non-blocking processing. Tomcat 8.5 implements Servlet 3.1 which supports non-blocking I/O. The prototypes were implemented with reactive programming and can thereby make use of this feature.

The results showed that the reactive prototype with the execution context in the thread which invoked the operator `subscribe` performed best. The prototype did not differ itself from the other prototypes in CPU usage and memory usage, but it performed better by having lower latency and higher throughput during high loads. The prototype increased the performance of the content server compared to the synchronous solution.

A possible explanation for the limited performance of the reactive prototypes with execution context in an elastic thread pool or in a fixed thread pool could be the different execution context. Changing threads are resource consuming and if tasks have very low latency this might decrease performance. In the case of the content server the processing of data is done asynchronously in one thread per request which makes it unnecessary to change execution context. It would be different if lists of data were to be processed. Then the changing of the execution context might have been suitable because a thread could be assigned of processing for a single data. In that case it is worth to consider to create a new thread for each single data or to use one or several thread pools.

Unfortunately the time was not enough to measure the latency impact for the whole back-end. This is something which can extend the research conducted in this master's thesis.

The conclusion drawn from the results is that reactive programming has a positive impact on the performance of the content server, but the impact is dependent on the execution context.

7.1.1 Importance of execution context

The execution context played a big role in the performance of the reactive prototypes. In each case where reactive programming is applied the decision of execution context needs to be evaluated because the impact of performance might not be the same as for the cases where reactive programming was applied in this master's thesis.

Research into execution context in a pull base solution would also be noteworthy because of the processing of lists. Due to the time restraint of this master's thesis this research was not pursued.

7.1.2 Memory

The memory measurements are difficult to draw any conclusions from even though in Figure 6.2 some differences can be seen. The figure shows that less memory is used by the reactive prototypes but this is the mean memory use from several tests. In reality the diversity in the measurements were vast. The memory usage for the synchronous solution could sometimes be less than for a reactive prototype, but the measurement proved that in general it was actually higher. Hence, it is hard to draw any conclusions from the results of the memory usage. What can be seen is that the stabilization of memory usage is faster for the reactive solutions while the memory usage takes longer time for the synchronous solution to stabilize. A possible conclusion could be that if the pressure from the content provider increases the memory usage will also increase but it takes longer time to decrease for the synchronous solution. Plotting the memory usage for each test shows that the usage follows a saw-tooth pattern because of the garbage collector. Because of the saw-tooth pattern it could also be argued that the mean memory usage is not important but rather the mean maximum memory usage is significant.

7.1.3 CPU

The results points to an equal CPU usage for all solutions even though they still performed differently. The results lead to the conclusion that reactive programming does not affect the CPU usage but rather the utilization of the CPU and this is based on the fact that the latency and throughput differed despite equal CPU usage.

The CPU measurements were constant between execution of tests and are therefore reliable to draw conclusions from. CPU spikes of few percents from time to time during the tests. The spikes were discarded as normal background work for the application because these spikes repeatedly happened during each test for every solution. These spikes and the

magnitude of them possibly contributed to the small differences in CPU usage between the solutions.

During the tests the CPU usage never rose above 7 percent. This can have impacted the results by not bringing out the performance differences to its full extent. The performance differences would probably have been clearer if the system was not mostly IDLE but instead performing work. The reason for not pushing the system further was how the tests were performed. A more efficient content provider would have been needed. The content provider used in the test could not push more data than this. Alternatively more instances of the content provider could have been run.

7.1.4 Latency

In Figures 6.3, 6.4, 6.6 and 6.7 the measurements for the latency is presented. All the prototypes performed unexpectedly regarding latency. One would expect that the latency increases or stays at the same level. However, the prototypes surprisingly showed lower latency during high load than during low load. One possible reason for this decrease in latency could be that the application was not pushed to use more CPU. While the other three prototypes showed an increase in latency under the highest load the mono prototype decreased in latency. The decrease of latency is also connected with the fact that this prototype had a hundred percent throughput.

Other reasons for these latency results regarding all the prototypes could also be the threading model of Tomcat and Spring, but the results could also be impacted by the JVM and the garbage collector.

7.1.5 When to apply Reactive Programming

From the results it can be concluded that reactive programming only affects the content server's performance during high load. In lower loads the content server solutions perform equally with differences in memory usage. Reactive programming could be a good choice because of the content server often working during high loads. For applications working with lower loads the performance gain compared to other aspects of reactive programming needs to be assessed.

In this master's thesis reactive programming has been applied to Java, which is a language built for imperative programming. Moving the paradigm from imperative to declarative the library Reactor was used. By using a programming language which was actually designed for declarative programming the results might have differed. The impact of reactive programming seen in this master's thesis should be considered with this in mind.

7.1.6 Impact of design choices

The design choices made for the implementation of reactive prototypes could have most definitely affected the results. Several other reactive libraries could have been used or even other methods of implementing reactive programming could have been chosen. These choices are worth keeping in mind for the future in order to find a suitable solution. For this project I found these design choices to be suitable.

Spring and Tomcat were used in all prototypes because it was also used in the synchronous solution, but these variables are not investigated. However, the choice of framework and application server can have an impact on the performance[25].

The choice of using a push model impacted the design choices. The choice of using the Reactor class `Mono` was solely based on the push model. The model locked the content server to handle pushes of single data. It would also be interesting to see how reactive programming would affect a content server using a pull model and probably the Reactor class `Flux` would be a better choice in that case. Parallel execution can be used when using a flux. This would be interesting to research because of Amdahl's law which predicts the theoretical speed up of a parallel system[9].

A possible improvement could be to execute the publishing of the data in different threads because the next callback method is not dependent on the publishing. The two tasks can even be processed in parallel.

7.1.7 Measurements

The CPU usage and memory usage was measured with the tool JVisualVM. Other tools exist to measure these metrics and these could have been used as well. Measuring the CPU usage gave consistent results but the measurements of the memory usage were inconsistent. To get more consistent measurements of memory usage the running time or in fact the execution of the test might have had to be changed. An alternative way to measure the memory is to keep track of the garbage collector[25].

Logging was only used during the tests measuring time to ensure as little impact on results from logging as possible. The logging was also done asynchronously to limit the time effects on the measurements. The results could still have been affected but the effect should be minimal.

7.2 Development Process

Implementing the content server using reactive programming did not change the development process significantly. The number of lines of code written did not differ substantially between the solutions which means that the complexity is rather the main factor to take into consideration.

The only part which could add complexity is the implementation of the handler methods because all other code was practically the same. Reactive programming did not add much complexity in this case since the processing stages are still the same as in the synchronous solution. The readability of the code possibly decreased since the kind of data that is emitted to next operator cannot be seen in the code. To find the type one must check the return type of the previous method invoked by an operator.

Chapter 8

Conclusion

During the course of this master's thesis a study was carried out about reactive programming. A synchronous solution of the content server was implemented and compared to reactive prototypes. The solutions were compared in performance and development process. The research questions presented in Section 1.3 have been researched and answered.

The results showed that the CPU usage did not differ between the solutions. The measurements of the memory usage were inconsistent and therefore it was difficult to draw a conclusion. The memory usage stabilized quicker than for the reactive prototypes for the synchronous solution. One solution stood out when comparing the latency and the throughput. That solution was the reactive prototype which executed the asynchronous task in the thread that invoked the operator `subscribe`. The difference could be seen when the content provider pushed data with 5 milliseconds intervals. At this interval the prototype had less latency and 100% throughput. Reactive programming can definitely have an effect on the performance of a content server application. However, the execution context of the asynchronous tasks, which would need to be further investigated, could highly influence the effect.

The development process itself was not significantly impacted by reactive programming. It was basically a reimplementation of the necessary handler methods. However, reactive programming added complexity to the code and it also required comprehensive knowledge on the execution context and on the reactive library.

To decide if reactive programming would be suitable to use in a project both the possible performance impact and the developer process needs to be taken into consideration. The performance was increased for the content server but reactive programming might have other effects on different applications and programming languages. That is also good to note that the programming language Java, which is designed as an imperative language, was used in this master's thesis and therefor the results could differ from solutions using declarative languages.

8.1 Future Work

In this master's thesis reactive programming was applied to the logic inside an application. The research raised many questions about areas which could not be covered in this master's thesis. These areas would need further research to provide better understanding of reactive programming.

Research into how reactive microservices would affect the performance of a content server application is of great interest. Splitting up a monolith into microservices can possibly affect the performance and facilitate horizontal scaling. Not only the performance but also the development process and research on scaling would be of interest.

It was shown that the execution context plays a big part in the performance when reactive programming is used. A comprehensive research on the most suitable execution context could also prove to be noteworthy. The choice of execution context might affect parts of an application differently.

During the development process several design choices were made. Changing any of these might have changed the performance of the application. The application can for example be developed with other reactive libraries or other solutions for example `CompletableFuture`.

The research in this master's thesis was carried out based on the use of a push model. Because of the push model the data was needed to be processed one-by-one and thereby a mono was used. Last but not least, it would also be interesting to research how reactive programming would affect a pull model or more generally how the processing of lists of data affects a content server. If a list is processed by using the Reactor class `Flux` than parallel execution can be achieved by using the operators `parallel` and `subscribeOn`.

Bibliography

[1] Apache Tomcat 8 Configuration Reference. https://tomcat.apache.org/tomcat-8.5-doc/config/http.html. Accessed: 2017-07-02.

[2] jdk9 Candidate classes Flow and SubmissionPublisher. http://cs.oswego.edu/pipermail/concurrency-interest/2015-January/013641.html. Accessed: 2017-06-06.

[3] JVisualVM. https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jvisualvm.html. Accessed: 2017-05-10.

[4] Processes and Threads. https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html. Accessed: 2017-06-06.

[5] Project Reactor. https://projectreactor.io/. Accessed: 2017-06-19.

[6] Reactive Streams. http://www.reactive-streams.org/. Accessed: 2017-06-06.

[7] ReactiveX. http://reactivex.io/. Accessed: 2017-06-09.

[8] The Reactive Manifesto. http://www.reactivemanifesto.org/. Accessed: 2017-06-06.

[9] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[10] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4):52, 2013.

[11] Boris Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold Co., 1984.

[12] Gérard Berry. Real time programming : special purpose or general purpose languages, 1989.

[13] Conal Elliott. The essence and origins of FRP. `http://conal.net/talks/essence-and-origins-of-frp-bayhac-2015.pdf`. Accessed: 2017-06-19.

[14] Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, number 8 in 32, pages 263–273, 1997.

[15] Conal M Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 25–36, 2009.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[17] David Harel and Amir Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498. Springer, 1985.

[18] Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, page 3, 2013.

[19] Frank Keller, Subahshini Gunasekharan, Neil Mayo, and Martin Corley. Timing accuracy of web experiments: A case study using the WebExp software package. *Behavior research methods*, 41(1):1–12, 2009.

[20] Samuel Kounev and Alejandro Buchmann. Improving Data Access of J2EE Applications by Exploiting Asynchronous Messaging and Caching Services. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 574–585. VLDB Endowment, 2002.

[21] Daniel A Menascé. Load testing of web sites. *IEEE Internet Computing*, 6(4):70–74, 2002.

[22] Bhoomit Patel, Jay Parikh, and Rushabh Shah. A Review Paper on Comparison of SQL Performance Analyzer Tools: Apache JMeter and HP LoadRunner. *International Journal of Current Engineering and Technology*, 4(5):3642–3645, 2014.

[23] Douglas C Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65–74, 1995.

[24] Don Syme, Tomas Petricek, and Dmitry Lomov. The F# asynchronous programming model. In *International Symposium on Practical Aspects of Declarative Languages*, pages 175–189, 2011.

[25] Konstantinos Vandikas and Vlasios Tsiatsis. Microservices in IoT clouds. In *Cloudification of the Internet of Things (CIoT)*, pages 1–6, 2016.

EXAMENSARBETE Reactive programming and its effect on performance and the development process
STUDENT Gustav Hochbergs
HANDLEDARE Patrik Persson (LTH), Johan Frick (Playtech BGT Sports)
EXAMINATOR Jörn Janneck (LTH)

Nå ut med information snabbare

POPULÄRVETENSKAPLIG SAMMANFATTNING **Gustav Hochbergs**

Idag förväntar sig alla ha den senaste informationen och att uppdateringar sker omedelbart. För att möta dessa krav testades reaktiv programmering. Detta medför att genomströmmningen av information ökar, trots att lika mycket datorkraft används.

Digitalisering har idag nått nya höjder och det finns en strävan att digitalisera mer och mer i samhället. Kraven på dessa digitaliserade tjänster är höga och kräver ett bra sätt att presentera information samt en välfungerande bearbetning av data. Mängden information som ska bearbetas har ökat och samtidigt finns förväntningarna att det är den senaste informationen som presenteras. Exempelvis så förväntar sig användarna av en ”betting-sida” att det är de senaste oddsen som presenteras. Detta för att användarna ska kunna ta ett beslut om att lägga ett ”bet”.

För att uppfylla användarnas förväntningar på tjänsterna behöver bearbetningen av data vara effektiv. Detta kan betyda att andra programmeringstekniker behöver användas. Reaktiv programmering är en teknik som kan liknas vid dagens hemleveranser av matkassar. Tiden det tar att åka till mataffären och handla kan istället användas för andra uppgifter genom att mataffären packar och levererar varorna till dig. Reaktiv programmering handlar om att inte låta tid och kraft gå åt till att vänta, utan använda denna tid och kraft till att göra andra uppgifter. När varorna är leverarad ”reagerar” du genom att laga mat medan de digitala tjänsterna istället reagerar på att data är tillgänglig genom att t.ex. uppdatera odds.

Detta arbete visar hur en ”betting-applikation” påverkadas av att applicera reaktiv programmering på bearbetningen av data. Genom att göra detta kan man se en ökning av genomströmmning när mycket data ska bearbetas. Genomströmmning är ett mått på hur mycket data som applikationen hinner bearbeta under en viss tid. Den reaktiva tekniken har vid bearbetning av mycket data en genomströmmning av 100%, vilket betyder att applikationen hinner bearbeta all data den ska. Resultatet kan jämföras med en genomströmmning av 80% när den reaktiva tekniken inte används. Intressant är att båda teknikerna använder lika mycket processorkraft, vilket betyder att reaktiv programmering bearbetar datan på ett mer effektivt sätt.

Datorkraften kan vara begränsande både av ekonomiska och fysiska skäl. För att uppfylla användarens förväntningar på att ha den senaste informationen kan istället en mer effektiv användning av den existerande datorkraften krävas.

Resultaten är inte enbart intressant för ”betting-applikationer”. De är även intressanta för applikationer som behöver förbättra prestandan när begränsad hårdvara finns tillgänglig.