Using Event- and Actor-driven Paradigms to Increase Web Server Performance

Felix Hessenberger

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2014

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 30, 2014

Felix Hessenberger

Contents

Abstract

Over the past years, websites have advanced from merely displaying content to representing interfaces to dynamic server-side applications of various scales; other environments like mobile platforms tend to use the same HTTP interfaces as well. To limit the cost of server hardware various software-based approaches aim to maximise the number of simultaneous operations by shifting from the classic per-request threading model to more sophisticated concurrency patterns. This thesis presents and compares a number of different approaches to server-side concurrency implementations from the view of a programmer. Typical use-cases for server-side information flow are contrived and evaluated regarding asynchronous processing. Patterns are then reviewed based on their performance in scalable high-throughput networking applications by the example of live applications as well as experimental settings.

Kurzfassung

Franz rast im komplett verwahrlosten Taxi quer durch Bayern.

Chapter 1

Introduction

1.1 Motivation

a

a

1.2 Objective

a

Not a comparison between threads and events Not for general programming, specific to Web

How and how much events and actors can increase performance

1.3 Structure

Chapter 2

Technical Background

2.1 Terms and Definitions

A Web server can be utilised to handle rather different tasks, from merely delivering static assets like images to serving entire Web pages to representing an endpoint for raw data retrieval. This section aims to give an overview of the basic requirements a modern Web server architecture needs to fulfil. Moreover, important performance factors are elaborated with regard on high-demand and high-performance setups.

2.1.1 Network Communication

The eponymous task of a Web server is to serve Web-connected clients over the medium of the Internet. This involves receiving and sending messages using different implementations of network protocols. The most widely used protocol of the Web, *HTTP*[1], is a request-response protocol, which means that for every message a client sends to a server, a response is sent back [31]. To minimise networking latency, it is preferable for a Web server to have a high-speed connection to the Internet, fast system I/O[2] and capable routing hardware. However, these parameters are not directly related to software and are thus neglected during the further course of this thesis.

2.1.2 Dynamic Content

Originally, the Web was intended to be a network of interconnected text files, which later was augmented with images and style sheets; Web servers were basically required to understand incoming requests and respond with static

[1] Hypertext Transfer Protocol
[2] Input and Output, esp. hardware

content accordingly [31]. With the release of *PHP*[3], *ASP*[4] and *Java*[5] – in 1995, 1996 and 1997, respectively – dynamic webpages, i.e. views that are prepared by the server based on dynamic data like database content, became widespread [42]. From that point on, Web servers needed more processing capabilities for script execution and database access; however, the number of requests remained roughly the same, except for occasional form submissions [42].

2.1.3 Asynchronous Requests

The advent of *AJAX*[6] and mobile applications in the late 2000's changed requirements drastically. Rather than refreshing the whole view for every piece of information sent and received, data could now be transferred in a more granular fashion. By asynchronously communicating with an API[7] endpoint in the background, operations like deleting an item from a list could be performed invisibly and ubiquitously without reloading the page context. Especially applications that aim to provide desktop-like capabilities – commonly called Rich Internet Applications – make heavy use of asynchronous requests [24, p. 4]. This inherently also changed users' expectations for websites from anticipating a certain amount of load time to implicating real-time behaviour [32]. To achieve low latency while maintaining client-server information consistency, the server's performance has to meet the combined request frequency of all clients at a given point in time.

2.1.4 Request Frequency and Response Time

Since in many cases the responsiveness of the user interface depends on the duration of the server communication roundtrip, maintaining acceptable response times is often crucial [21, p. 1]. Request frequency and response time correlate in the sense that request frequency represents the demand on a server endpoint while response time – given equally demanding operations per request – can be interpreted as the potential of the server to meet the demand. When the processing limit of the server is met, response times become generally inversely proportional to the request frequency, as illustrated in figure 5.7 [39]. At this point, the server may chose to neglect the request (ideally by returning the status code *503 Service Unavailable* [31]), not respond at all or even stop serving clients altogether (i.e. "crash").

[3]Recursive acronym: PHP Hypertext Preprocessor, http://php.net/
[4]Active Server Pages, http://msdn.microsoft.com/en-us/library/aa286483.aspx
[5]https://www.java.com/
[6]Asynchrounous JavaScript and XML (Extensible Markup Language)
[7]Application Programming Interfaces

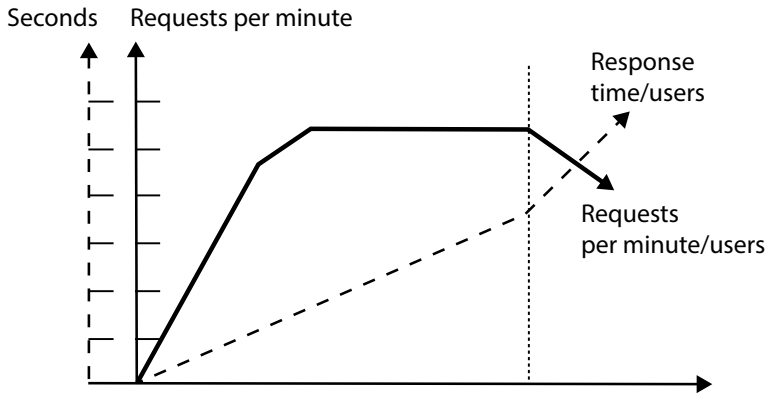

Figure 2.1: Correlation between request frequency and response time in a typical Web server setup. After the server has reached its limit of linearly serving clients (indicated by the dotted line), response times become inversely proportional to the request frequency. Image source: [39]

2.1.5 Scalability

Demands on Web servers typically are lower during the initial phase of a business and grow with the popularity of the service. Since business growth and server load can not be exactly predicted, it is necessary to be able to adjust (i.e. *scale*) the entire server architecture according to current needs in a timely manner. The *Slashdot Effect* describes a sudden spike in service popularity and can – due to the open nature of the Web – lead to a tremendous increase in activity over a relatively short timespan [6, p. 1].

Today's hardware is well suited to meet high demands and can be configured flexibly: If a larger number of physical server units as well as the necessary infrastructure is available, requests can be distributed and the load on a single unit decreases. If single units are outfitted with more memory and faster processors, the number of request operations per unit increases. Since acquiring and maintaining server units and other infrastructure components is expensive, well-designed software can make a significant difference in system efficiency, which in turn can greatly benefit any business – especially with modern service providers*pay-what-you-use*[8] [18, p. 11]. Software that is well-suited to be expanded according to its usage is considered *elastic* [3].

Ideally, the server software should be hardware agnostic, i.e. should behave consistently independent of the hardware it runs on. For instance, if the software depends heavily on sharing application state via RAM[9], scaling out on more than one machine will be unsuccessful [27]. Scalability can be measured by the relationship between hardware resources and the increase

[8]So-called "Cloud"-services provide flexible plans on processing power, that often can be dynamically adjusted without significant server down-time.

[9]Random Access Memory

of performance. If this relationship is nearly linear, the system can be considered to scale well.

2.1.6 Development

Not a part of the production system itself, but nonetheless an essential part of all Web server applications is their development. A structured, idiomatic way of writing application logic doubtlessly contributes to every software product. Modularisation of components facilitate the use of third-party software like libraries and frameworks. In return, using existing solutions can greatly reduce development time and effort, while simultaneously providing proven solutions. Web server applications particularly benefit from frameworks since they often handle standard tasks like network I/O^2, database access and caching [23, Foreword]. Integrating and maintaining these frameworks is a major part in implementing a Web server application; thus, not only the performance, but also the ease of use of selected frameworks and their language environments by the developer are treated as criteria in this thesis.

2.2 Concurrency Models

Since a Web application in a production setting is usually publicly accessible, serving multiple clients simultaneously is the rule, rather than the exception. Depending on the popularity of the service, the number of concurrent requests can range anywhere from dozens to several thousands, e.g. for social media sites [6, p. 1]. A server process with a single flow of control would only be able to serve one client at once, with all requests received while the server is busy being neglected. Therefore, networking applications always have to be implemented using multiple program flows that can be executed concurrently [46]. This section lists various paradigms associated with designing an application capable of maintaining multiple flows of control.

2.2.1 Purely Thread-based Model

A thread is a sequence of instructions within a program. Allocating processing time to threads is handled by an operating system scheduler. To have a program execute multiple logic structures concurrently, they have to be explicitly abstracted in the form of threads. Physical concurrency occurs, when threads are executed simultaneously – i.e. at the exactly same time – on different processor cores; in contrast, logical concurrency describes that multiple threads are executed sequentially in rapid succession at roughly the same time, thus giving the impression of simultaneous execution. Physical concurrency is inherently more efficient [38].

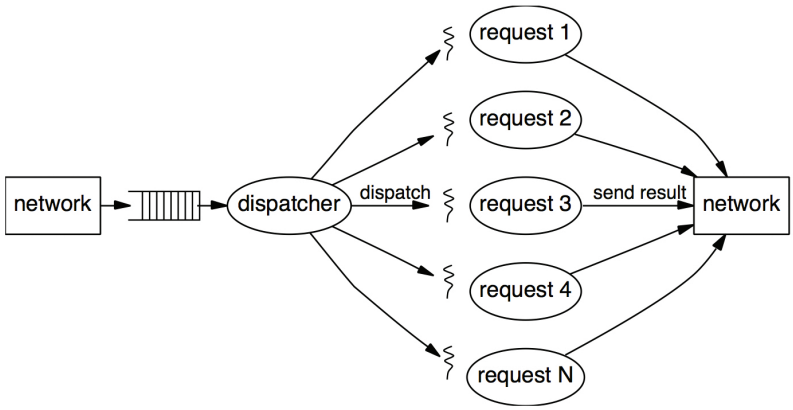

Figure 2.2: On a purely thread-based Web server, each request is handled by a dedicated thread. Incoming network requests are queued and sequentially accepted by a dispatcher, that distributes them among available threads – either by using idle threads from a thread pool or by creating new threads. Image source: [28]

Flow of Control

A great advantage of threads in the context of Web server applications lies in the natural abstraction level regarding multiple parallel requests: Client communication is commonly treated as a set of mutually independent connections; this approach of abstraction facilitates a clear program flow structure [27]. According to this model, every request can be treated as an isolated flow of control (see figure 2.2). However, since threads are not isolated from each other and share state via a common memory address space, this only holds true as long as resources like queues or caches are accessed sequentially [2, p. 2]. Thus, developers have to pay close attention to avoid race-conditions, deadlocks and access violations – complications that generally result from improper thread coordination [9, p. 1]. Therefore, the implementation of large-scale systems heavily relying on threads always introduces additional complexity [19, p. 1].

Scalability

Traditionally, Web server applications process each request on a dedicated thread throughout its whole lifespan, from accepting it to responding to it [15, p. 162]. This behaviour can be observed for instance in implementations of the popular LAMP[10] server stack configuration [15, p. 48]. A less experienced programmer might find this ideal, since concurrency stays mostly

[10]Linux, Apache, MySQL, PHP

hidden and the application logic is orientated on the flow of a single request – smaller projects might not experience any drawbacks of this setup at all. However, it is obvious that to scale up a thread-based system the number of threads has to be increased. The number of threads engaging in simultaneous processing, i.e. physical concurrency, is limited by the number of processing cores. This means that on a computer equipped with a quad-core processor, four threads can be executed – and thus, four requests can be served – in parallel[11].

Drawbacks

Problems arise when a thread has to wait for another requirement to be fulfilled. The process of meeting a requirement that renders the executing thread unable to proceed is called a *blocking* operation. Such actions include for instance reading or writing a file on mass storage, handling network traffic or file uploads, querying a database, accessing another Web service or doing intensive computations [15, p. 196]. When a thread encounters a blocking operation, it cannot advance further in the program flow until the operation completes, as illustrated in figure 2.3. The resulting delay can account to anywhere from a few milliseconds to several seconds, for instance when accessing a slow or irresponsive Web service. The only way to counteract the temporary occupation of threads and to continue processing incoming requests is the creation of new threads [18, p. 36]. However, every newly created thread counts towards certain limitations in scalability. On the one hand every thread receives a predefined share of process address space memory – also known as *stack* – upon creation to temporarily store data [43]; since memory is reserved in advance without knowing the exact requirements of the thread, a certain amount of memory overhead is likely. On the other hand, the entirety of all threads has to be orchestrated by an operating system module called *scheduler*, which requires processing time relative to the number of threads [43]. Moreover, a computationally expensive procedure called *context switching* must also be followed upon changing the actively processed thread [37]. This process includes complications called *buffer* and *cache misses* as well as lock contentions [28, p. 2]. When a certain number of active threads is reached, this can lead to serious performance degradation, as illustrated in figure 2.4. Especially when the application is executed inside a *virtual machine*[12] – which is often the case for distributed applications due to better replicability – the over-provisioning of memory leads to scarce resources [12, p. 1].

[11] Certain implementations of simultaneous multithreading allow for increasing this number at the cost of reduced performance per thread, for instance Intel's Hyper-Threading Technology (http://www.intel.com/).

[12] A software-based emulation of a computer, that executes programs like a physical machine.

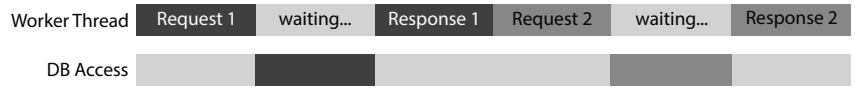

Figure 2.3: A typical blocking situation in Web server scenario. When request one (shown in dark grey) arrives, a database operation is necessary. During the course of this operation, the executing thread blocks while waiting for results. The response can only be sent when data is returned and the next request (shown in medium grey) can only be served after the first one completes.

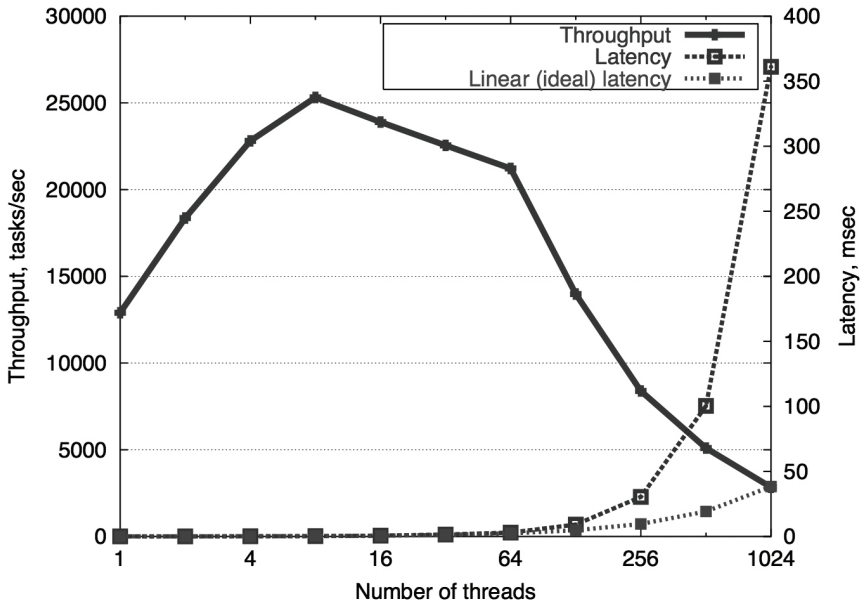

Figure 2.4: This graph shows the performance degradation resulting from rising request frequency for a purely thread-based Web server. Because of the performance overhead introduced by a large number of concurrent threads, the processing throughput decreases. If the number of threads grows very large, the request-response latency escalates due to the shortage of resources. The data is taken from an experiment by M. Welch et al. [28]. Image source: [28]

Some of the problems of threads can be addressed by using a *thread pool*: Instead of spawning new threads upon each request, a fixed number of threads is spawned in advanced and workload is distributed among them. However, this procedure is not without problems and introduces the delicate step of setting the thread pool size [33]. It can be concluded that a lower thread overhead can benefit the overall performance of a process. Furthermore, when scaling an application, the maximum number of simultaneously

processed threads can at best only increase linearly in relation to the number of processing cores in a system [36].

2.2.2 Event-based Model

While threads on their own present a convenient abstraction for handling Web requests, recent years have seen an incline towards event-driven architectures [25]. Events can be seen as an arbitrary change in application state; an example of an event may be an arriving HTTP request. An event is often abstracted to an object that is passed along with the flow of control and may consist of a header describing its nature (e.g. the fact that it represents an HTTP request) and a body containing additional information (e.g. request parameters and client identification). A different part of the application may subsequently *react* to this event by executing further operations like querying the database.

Using events is a significant departure from the traditional command-and-control style used for instance in purely thread-based architectures (see section 2.2.1). However, seen from a different perspective, using events on a Web server is at least as idiomatic as using threads: The Web server has no control over the arriving requests, yet it has to respond by executing application logic. Instead of forcefully maintaining control over the execution context, the Web server may relinquish control and let itself be controlled by events. This strategy follows the principle of *inversion of control* [17].

Relation to Threads

Event-driven programming does not preclude the existence of threads; neither is it the opposite or an evolutionary step. All major operating systems use threads as a means of managing process execution; thus, even a purely event-driven program runs at least on one thread.

Flow of Control

While event-driven programming does not imply a certain concurrency model, the employed concepts have a strong influence on how concurrency is handles by the application. This section elaborates the basic concepts of event-driven programming.

At its simplest, an event-driven application consists of two major components: On the one hand an *event loop* containing an *event listener* and on the other hand an *event handler*. The event loop is a lightweight structure passing incoming events from a queue to event listeners that have subscribed to a certain kind of event, e.g. an incoming network request. The targeted event listener then passes the event on to a handler function, which executes application logic and may create another event upon completion. Larger applications typically have one event loop per process and a number of listeners

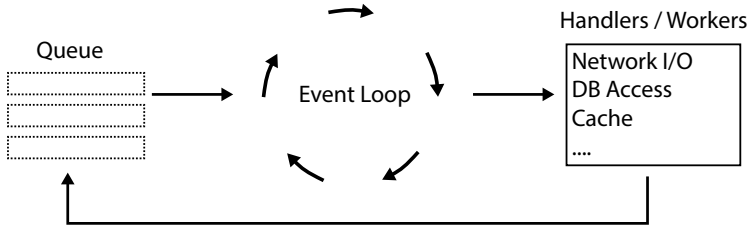

Figure 2.5: Basic flow of control in an event-driven application. Operations that would normally block the event loop are executed separately and create further events upon completion.

and handlers [18, p. 33]. For an illustration of a basic event-driven application setup, see figure 2.5.

The use of events leads to an inherently *flat* application structure in the sense that there is no hierarchical ordering of event sources and destinations. There are two ways of advancing in the flow of control at the end of a particular operation: The first option is to create a new event that is received by the event loop and propagated to the next event handler. The second way is to employ a *callback function*. Calling a callback function can be regarded as transferring the flow of control to an event handler without consulting the event loop [7, p. 92]. Callback functions are often used to handle results of a blocking operation – like making a request to a remote Web service – directly upon its completion and thus maintaining control over the execution order of further actions. Implementation is typically done via either a function pointer or an *annonymous function*, as demonstrated in program 2.1.

Departure From the Call Stack

Neither events nor callback functions are usually found in traditional – i.e. sequential – programming. In nearly all application environments the default way of executing routines in succession is to use functions, whose return values are used in the further course of the program. This proven concept determines three major aspects of program flow [17, p. 3]:

Coordination The ordered execution of sequential operations, which prevents problems associated with concurrency

Continuation The program flow continues immediately after the function call, thus eliminating the need to explicitly define a continuation point

Context The proper handling of the local variable scope; if a function returns, the previous context is restored and the callee function can use the same variable scope as before the calling operation

Program 2.1: Calling a callback function via a function pointer (above) and via an annonymous function (below) in JavaScript. The request to a Web service may take some time and is thus executed asynchronously. When the response from the service arrives, the callback is executed. For this example, the response is printed to the console, which is a rather fast action and therefore can be executed in a blocking fashion.

```
1 function callbackFunction (data) {
2     console.log(data);
3 }
4
5 WebService.get("http://example.com/").then(callbackFunction);
6
```

```
1 WebService.get("http://example.com/").then(function(data) {
2     console.log(data);
3 });
4
```

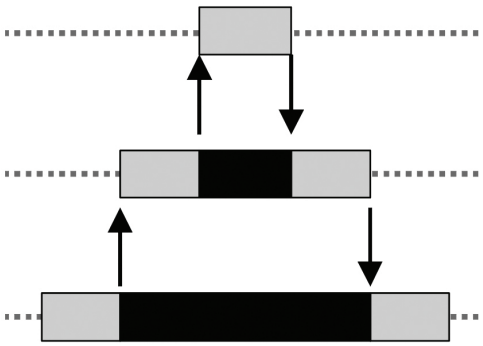

Figure 2.6: Illustration of a simple call stack structure. As time progresses (horizontal axis, left to right), the call stack grows with each function call. As the functions return gradually, the stack size decreases again. Active parts of functions are shown in grey, passive (i.e. *waiting*) parts are shown in black. Image source: [17]

To store the references and contexts of all functions during such a succession of calls, a dedicated part of memory called the *call stack* is used. An illustration of a simple call stack is shown in figure 2.6.

Despite its advantages, heavy use of the stack in the context of modern Web applications poses two substantial problems: On the one hand, the

concept of the stack origins from a time where concurrent computing was not frequently used – especially in the manner seen in highly concurrent applications. The behaviour of the stack pursues a strongly linear manner of execution. Because at any point in time, only one call can be pushed onto the stack at once, only one action can happen at a time. Likewise, if an operation were to take an extended or unspecified period of time – like accessing a remote Web service – a single call stack provides no way of executing another operation during this time. The second problem of the call stack is that it can not be distributed across physically or logically separated systems. Pushing a call onto the stack implies that a return memory address – i.e. a continuation point – is known and clearly specified, which is not the case for distributed systems [17].

The departure from the stack also implies a concept called *loose coupling*. This means, that the components of interaction within a program do not need to know the exact specifications of the target. One example would be a new user registering for the service: A tightly coupled system would need to call the exact function that creates the user in the database. In an event-based system an event called *userCreated* (or similar) would be created and the database component would receive this event. This leads to a more flexible and resilient application structure, because changes to the exact implementation of one component do not require changes on the other side. The act of developing an application without a call stack is known as *stack ripping* [6].

Scalability

In contrast to the purely thread-based concept presented in section 2.2.1, event-based architectures tend to be better scalable. One major reason for this is the blocking behaviour of the I/O thread; while in purely thread-based systems the thread accepting an incoming request is no longer available for processing until all blocking operations have finished, the I/O thread in event-based systems only processes short-lived operations. Thus, in the latter, scalability is not directly proportional to the number of threads used by the system [5, p. 2]. A typical example of this behaviour is shown in figure 2.8. In this example, an event-based server can achieve a roughly 1000 times higher throughput per thread, than that of a purely thread-based one.

Furthermore, because the execution context of event-based program flow is event-specific, no global context switching has to be done. This leads to an increased actual concurrency of executed program code compared to purely thread-based systems [19].

To scale a single event loop on one machine, one event loop process can be created for every processing core – on a machine with a quad-core processor, four event loops can process incoming requests in parallel[11]. In such on-system scaling situations, event-based systems have the advantage, that

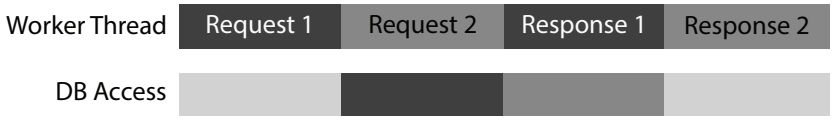

Figure 2.7: A non-blocking situation in Web server scenario. When request one (shown in dark grey) arrives, a database operation is necessary. This is a blocking operation, but since the worker thread does not have to wait for it to complete, the next request (shown in medium grey) can already be accepted and another database operation can be initiated. When the first database operation completes, the worker thread can send the response to the first client. Using more threads, this procedure can be heavily parallelised.

due to the reduced mutual exclusiveness of the program code, less locking situations occur. An example for this is, when a queue is accessed by multiple threads - only one thread can access the queue simultaneously, the other one has to wait.

Generally, it can be concluded, that for the specific scenario of a networking application like a Web server, event-based systems can provide more efficient performance and can thus be scaled more extensively with respect on minimal hardware requirements. Comparing figure 2.3 and figure 2.7, it can be seen that given a blocking scenario like a database operation, event-based concurrency not only benefits the number of parallel requests, but can also lead to significant response time improvements.

Drawbacks

Beside the implications of relinquishing the call stack pattern – like flat program structure, more or less obfuscated flow of control (see section 2.2.2, *Flow of Control*) and reduced state management capabilities (see section 2.2.2, *Departure From the Call Stack*) – there are other factors that have to be taken into account when using event-driven architectures:

Due to the non-linear nature of an event-based system, *race conditions* can occur. Race conditions typically happen, when the programmer expects a certain order of command execution, which are not guaranteed to be maintained under varying circumstances. For instance, if two Web requests are executed concurrently and the second response is expected to be always received after the first – because it is supposed to trigger application logic that depends on the first response's data – the application would fail if the responses would arrive out of order.

Additionally, event-based programs lack certain compiler[13] optimisations (given that the programming language used is compiled, rather than inter-

[13]Compiling is the process of transforming a human-readable programming language into code that is better suited for execution by computers, e.g. binary code.

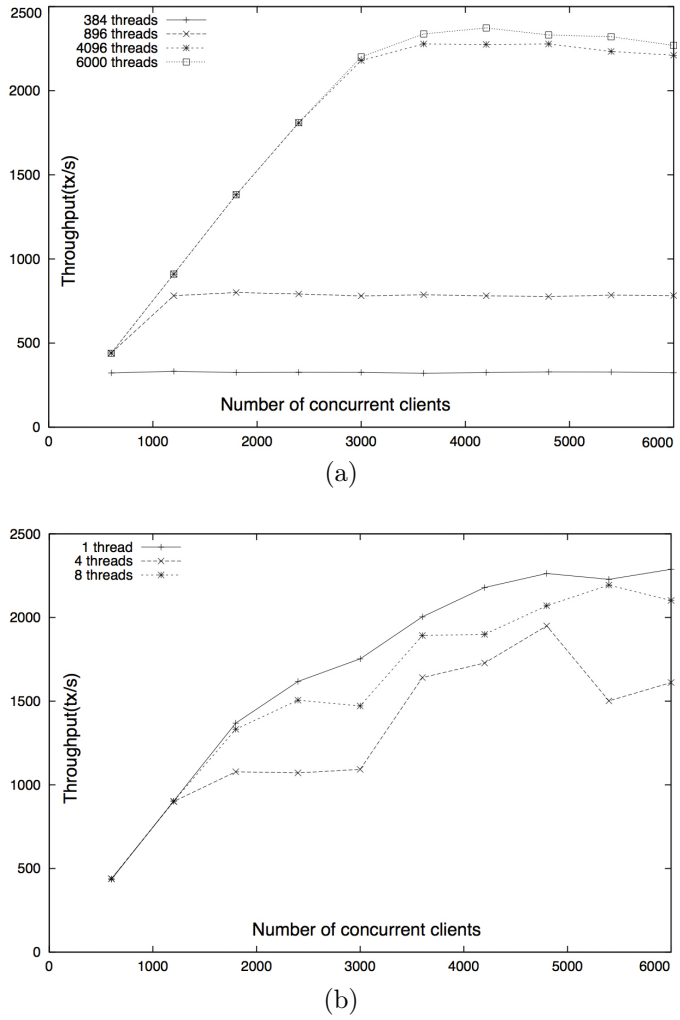

Figure 2.8: Typical scaling behaviour of a purely thread-based application versus an event-based application. This example is taken from an experiment by D. Carrera et al. [5]. The graphs show the effect of an increasing number of clients (x-axis) on the request throughput (y-axis) using a standard *httdp 2.0* Web server (http://httpd.apache.org/) – shown in (a) – and using *Java*'s event-based *NIO* interface (http://www.oracle.com/) – shown in (b). Image source: [5].

preted) like advanced memory management, inline functions[14] and compile-time warnings about race conditions [2, p. 5].

[14]Function content is directly inserted into the code instead of calling the function reference multiple times. This leads to reduced execution time and memory overhead.

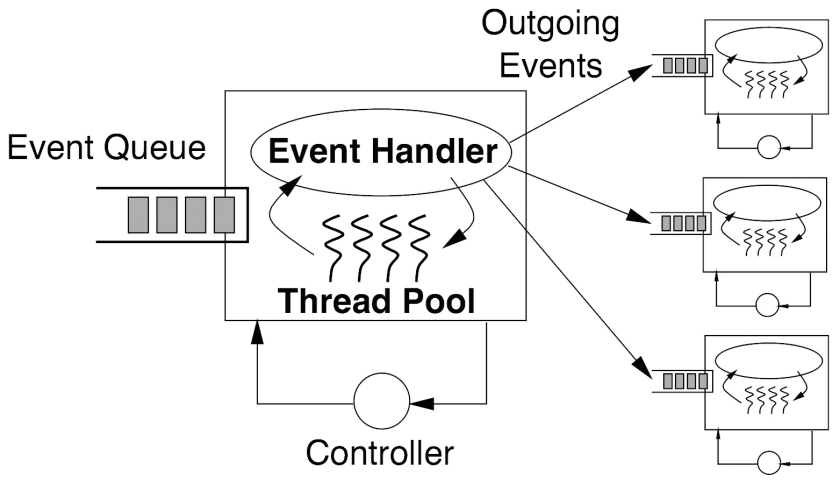

Figure 2.9: Illustration of a single SEDA stage (left) and the communication with other stages (right). Each stage has its own event queue, event handler, thread pool and resource controller. Image source: [28]

2.2.3 Staged Event-driven Architecture

Staged event-driven architectures (*SEDA*) describe a special variant of event-driven program flow that is rather different from the standard implementation (see section 2.2.2) in certain aspects and can be to some amount considered the middle ground between purely thread-based and event-based architectures. The eponymous extension to traditional event-driven architecture is the presence of *stages*, i.e. self-contained application components each including an event queue and a comparably small, dynamically-sized thread pool (see figure 2.9). Additionally, each stage has monitoring and controlling agents that enable introspection of the application. This way, parameters like the number of threads in the thread pool and the batch size – i.e. the number of events processed simultaneously in the stage – can be dynamically adjusted by the application [28].

Flow of Control

This variation of application flow moderately reduces the effects of inversion of control introduced by event-driven architecture; an inherent advantage of this design is the introduction of modularity and structured flow of control. Another advantage of SEDA is the adaptability to low-level operating system procedures like thread scheduling due to the aforementioned resource control elements; for instance, if less threads are available to a certain stage, the batch size can be reduced and thus the throughput can be maintained. Similar mechanisms can provide a certain level of explicit overload protec-

tion. An application that balances its own resources and makes decisions based on parameters like demand and response time is called a *conditioned* application [28].

Drawbacks

The coupling of stages by means of event queues is not necessarily supporting a clear application structure. M. Welsh, one of the developers mainly involved in the invention of SEDA in 1999, states this coupling as a main problem of SEDA [47]:

> If I were to design SEDA today, I would decouple stages (i.e., code modules) from queues and thread pools (i.e., concurrency boundaries). Stages are still useful as a structuring primitive, but it is probably best to group multiple stages within a single "thread pool domain" where latency is critical. Most stages should be connected via direct function call. I would only put a separate thread pool and queue in front of a group of stages that have long latency or nondeterministic runtime, such as performing disk I/O.

Furthermore, due to the fact that every stage has its own thread pool, context switching overhead is generally considerably higher than in traditional event-based architectures [47]. Apart from that, queues are considered an unsuitable data structure for high-concurrency applications, because they do not allow for concurrent access by multiple parties. Lastly, SEDA requires a fair amount of fine-tuning on the part of the developer in order to function flawlessly [41].

2.2.4 Actor Model

In an actor-based architecture, *actors* are the universal primitive of concurrent processing. An actor is an isolated (i.e. self-contained) entity that executes program logic. Actors can communicate with each other via asynchronous messages that contain arbitrary data and a reference to the sender. Based on the nature of the received message, the actor can execute side-effect-logic like writing a file to disk, but can also send messages to other actors – including the original sender. Actor-based architectures are similar to event-based architectures in the sense that both implement communication via message-passing. The concept of a direct response like in all aforementioned architectural patterns – be it via a call stack or a callback handler – becomes irrelevant [12].

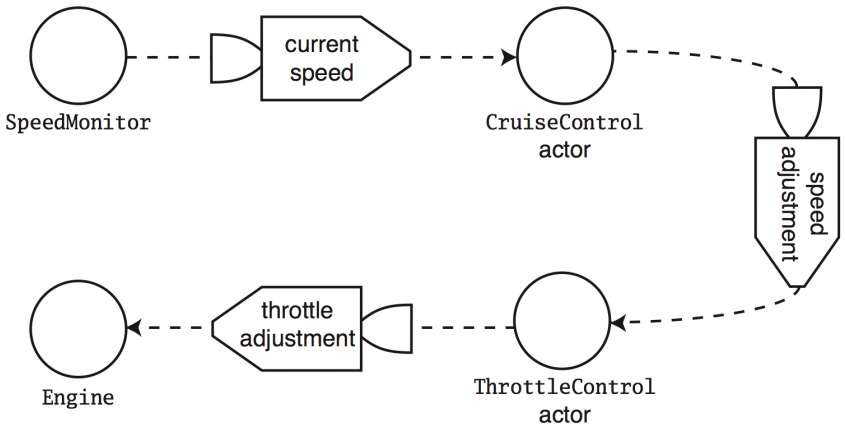

Figure 2.10: Illustration of an actor-based flow of control. Actors are represented as circles and messages are represented as rockets. The speed monitor actor periodically sends a message containing the current speed to the cruise control actor. If a deviation of the desired speed is detected, an adjustment message is sent to the throttle control actor, who again sends a message to the engine. Image source: [14]

Flow of Control

Event-based architectures work by the principle of *inversion of control* (see section 2.2.2), which is a necessity given the mechanisms of events. However, this limits the clearly defined structure of the program code and tends to be less idiomatic in nature. Message passing between actors, on the other hand, does not rely on inversion of control [12]. Control remains solely with the sending actor, since he can decide, to whom – if at all – to send further messages (see figure 2.10 for a functional diagram). This aids in reduced complexity when designing concurrent programs. Seen in this perspective, actor-based architectures can be seen as a balanced compromise between thread- and event-based architectures.

Another aspect of actor-based program flow is that the loose coupling introduced by events is further extended. Instead of memorising references to the callee function, actors have the choice of sending a message back; this introduces symmetry to logical communication. Furthermore, an actor's internal state can only change in response to incoming messages [14, p. 38]. This simple fact has great implications on state management across the application; this architectural characteristic is known as *share-nothing* architecture [4, p. 3].

Incoming messages are handled via *mailboxes*, which are basically actor-specific queues that receive, filter and defer incoming messages. Like in event-based architectures, there is no guarantee in which order messages arrive in

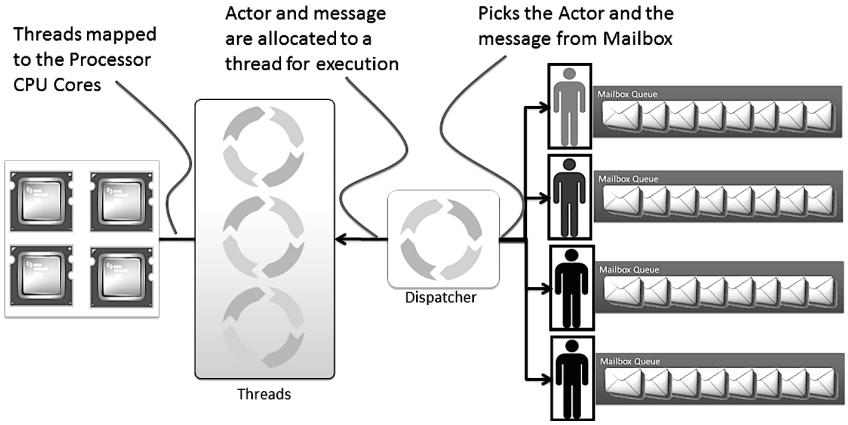

Figure 2.11: This graphic illustrates how resources are handled in a typical actor system. A *dispatcher* distributes messages of actors to operating system threads based on certain strategies (e.g. load balancing). Image source: [11]

the mailbox and – since the actor model does not specify a medium for passing messages (see section *Scalability*), a message can take arbitrarily long to reach its destination [7, p. 97]. Since only the corresponding actor can access his mailbox and only one message is processed at a time, no concurrency issues (like race conditions) can arise for the actor state and the mailbox as well as in between messages [8, p. 12]. The orchestration of message processing is done by a *dispatcher*, i.e. a part of framework logic that handles the execution of asynchronous, actor-based logic [11, p. 97]. See figure 3.3 for a basic illustration of resource mapping within an actor system.

Actors are also *resilient*, i.e. robust in the case of failure. *Erlang*, one of the programming languages first to embrace the actor model, coined the term *let-it-crash*; due to the isolated nature of actors, one actor unable to proceed working (i.e. *crashing*) will not affect other actors. The actor model also provides hierarchical structures of actors, which enable supervising actors to restart crashed actors [1].

Scalability

Due to the lightweight nature of actors, they can be used in abundance on single systems. P. Haller and M. Odersky state, that 5000 concurrently active threads can support over 1200000 concurrent actors [13, p. 2]. Also, due to their *share-nothing* nature, actors are not limited to one physical system, but work as well on distributed and replicated systems [11, p. 233]. This makes the actor model by far the most scalable of all listed herein.

Drawbacks

The practice of relinquishing shared state and using immutable messages for communication reduces the risk of problems inherent to high concurrency – like race conditions and deadlocks –, but does not eliminate them. Furthermore, inconveniences arise, when a specific execution order is crucial; while event-based architectures provide *callback functions* to handle this situation, guaranteeing execution order within actor-based architectures is non-trivial. What's more is that actors do not provide any means of object orientation or inheritance [20]. Also, like in event-based architectures, there is no additional compiler support (see section 2.2.2).

Chapter 3

State of the Art

As already mentioned in section 2.1, *Development*, developing software is greatly facilitated using existing building blocks instead of writing the entirety of program code from scratch. Especially Web server applications benefit from *frameworks* – i.e. third-party software that can be extended with application-specific code – because frameworks generally provide support for standard, repetitive procedures like handling network communication, database access, caching[1] and URL mapping[2]. The process of minimising repetitive code and maximising code reusability can be described as reducing *boilerplate code* or by the slogan *DRY*[3] [16, p. 149] [22, p. 1].

On the other hand, modern Web frameworks often include ways of abstracting concurrency or build on existing concurrency frameworks themselves. This can save the developer from having to deal with low-level concerns like thread scheduling and message passing (see section 2.2.1 and 2.2.4, respectively). *Full-stack* Web frameworks often handle many – if not all – tasks common to specific networking applications. They may even include their own Web server to improve the handling of numerous concurrent requests. Figure 3.1 gives a brief overview of typical full-stack framework capabilities.

This chapter presents selected approaches to performance-critical event-based and actor-based concurrency abstraction with respect to the criteria defined in section 2.1 and taking into account the technical issues elaborated in section 2.2.

[1] The *cache* is mainly short-lived memory used for faster delivery of dynamic data.

[2] URL mapping is the process of deciding which action should be taken based on the requested network URL.

[3] Don't repeat yourself

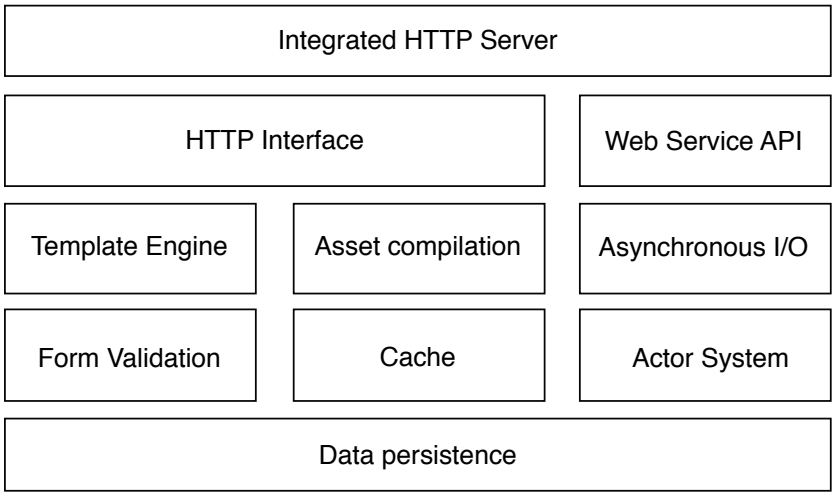

Figure 3.1: A Web framework aims to facilitate development by including frequently used capabilities. Inbound and outbound network communication represents a large share of common features (top three elements). Also, served assets like websites and images play a role in certain use-cases (but do not in case of a purely API-focussed server). Storage in form of cache and data persistance is also important for most applications. However, asynchronous I/O and Event- and Actor-based interfaces are the main focus of this chapter. Image based on the structure of the *Play! Framework* (http://www.playframework.com) [16]

3.1 Event-based technologies

3.1.1 Node.js

Node.js[4] is not only a framework, but rather a dedicated open source software platform for purely event-driven applications. User-level code is written in *JavaScript* and interpreted by the *V8*[5] engine also used in the *Google Chrome*[6] Web browser [18, p. 19]. This is reasonable since *JavaScript* was originally used primarily for programming client-side website behaviour; however, Node.js uses a module system[7] to add various Web server-specific features. These features include – among others – networking abstractions, file and operating system abstractions and replication and scheduling utilities[8]. The platform also includes its own Web server via the `http` module.

[4] http://nodejs.org/
[5] https://code.google.com/p/v8/
[6] https://www.google.com/intl/en/chrome/browser/
[7] Using the *CommonJS* module specification (http://www.commonjs.org/)
[8] See http://nodejs.org/api/ for an exhaustive list of system modules or https://www.npmjs.org/ for a popular extension module repository

Program 3.1: This example illustrates the concepts introduced at the beginning of section 3.1.1. In line 1, a HTTP network abstraction is loaded and exported to a variable for later use. Line 2 calls a function on this variable, requesting the creation of a new server instance; this function takes as parameter an *anonymous* callback function (meaning that it is passed directly in form of a function rather than as an assigned variable), which is called upon each incoming HTTP request. The function's two parameters are handles to the HTTP request and response, respectively. Line 3 and 4 generate the response by setting the HTTP status code, the `Content-Type` header and the response body. The server is started via the function `listen`, which accepts a network port and IP address. Code source: [18, p. 9]

```
1 var http = require('http');
2 http.createServer(function (req, res) {
3     res.writeHead(200, {'Content-Type': 'text/plain'});
4     res.end('Hello World\n');
5 }).listen(8124, "127.0.0.1");
6 console.log('Server running at http://127.0.0.1:8124/');
7
```

Development

Program modules are represented by *JavaScript* files. The entry point of a program must be defined by specifying the main *JavaScript* file upon application launch [18, p. 16]. To use a module it has to be included with the `require` command. Program flow typically propagates via callbacks (see section 2.2.2). An example of these concepts can be seen in program 3.1.

Due to the functional nature of *JavaScript*, callback functions can be the main driving force of asynchronous program flow. The following factors support this [45]:

- First-class functions can be handled like any other data type; they can be stored in variables, passed as parameters and executed when needed.
- Functions can be composed of multiple *anonymous* functions, which allows for flexible ordered execution (as seen in program 3.1).

However, there are several caveats to exclusively callback-driven program flow. For one, multiple callbacks executed sequentially are not guaranteed to return in a certain order. There is also no predefined way of awaiting multiple callback results. Also, callbacks – when used excessively – tend to lead to quite unreadable code and, ultimately, to a condition known as the *Pyramid of Doom* (see program 3.2).

To relieve these problems, the *promise* paradigm can be used as an abstraction for callbacks. With a promise framework like the one included in

Program 3.2: Multiple dependent callback functions can lead to a structure called *Pyramid of Doom*, which can impede code readability. Every step function (i.e. `step1`, `step2`, ...) asynchronously depends on the result of the previous one. In this example, code indentation tends to increase faster than line progression. Code source: [26, p. 21]

```
step1(function (value1) {
    step2(value1, function (value2) {
        step3(value2, function (value3) {
            step4(value3, function (value4) {
                // Do something with value4
            });
        });
    });
});

```

the popular *jQuery* library[9] or the *Q* library[10], sequential functions can be written more idiomatically as a chain of commands (see program 3.3). When a promise is created, the result is *deferred*, i.e. returned at a later point in time. If the action of the promise was successful, the promise is *resolved*, otherwise it is *rejected*. There is also a comprehension that allows for resolving multiple promises in parallel and treating the results as a single array of values as soon as all promises are resolved:

```
Q.all([stepA, stepB]).then(function (results) {
    var resultA = results[0];
    var resultB = results[1];
});
```

Another means of program flow in *JavaScript* is via explicit events. In *Node.js*, this can be conveniently done by using the `events` module. New events are created and handled by an instance of `EventEmitter` (see program 3.4). This way, a very flexible (yet *flat*, see section 2.2.2) program flow can be realised.

The `async`[11] module provides a number of functions that abstract and simplify working with asynchronous actions in a functional way and has an even wider scope than the *Q* library. For instance, it introduces comprehensions to apply an asynchronous function to multiple values (`each()`) and facilitates control flow with helpers for serial and parallel execution:

```
async.parallel([
    function(){ ... },
    function(){ ... }
], callback);
```

[9] http://jquery.com/
[10] https://github.com/kriskowal/q
[11] https://github.com/caolan/async

Program 3.3: By using a promise library, sequential asynchronous processing can be simplified. The `then` function accepts a first-class callback function and, optionally, an error handler (as seen in line 7). Code source: [26, p. 21]

```
1 step1()
2 .then(step2)
3 .then(step3)
4 .then(step4)
5 .then(function (value4) {
6     // Do something with value4
7 }, function (error) {
8     // Handle any error from step1 through step4
9 })
10
```

Program 3.4: A simple example of explicit events. First, the emitter created through the `events` module registers a behaviour (in form of a callback function) for a certain event type (i.e. `doorOpen`). At an arbitrary point in time an event of this type is created and triggers the callback function. Code source: [30]

```
1 var events = require('events');
2 var eventEmitter = new events.EventEmitter();
3
4 var ringBell = function ringBell()
5 {
6     console.log('ring ring ring');
7 }
8 eventEmitter.on('doorOpen', ringBell);
9
10 eventEmitter.emit('doorOpen');
11
```

Independent of the exact method of implementing concurrency in *Node.js*, *inversion of control* (see section 2.2.2) plays a big role and its drawbacks (e.g. reduced code readability) are hard to avoid without using special libraries [7, p. 93]. However, because *JavaScript* is a very popular language due to its use in website development, the adoption rate and the enthusiasm event of less experienced developers help with building a rich ecosystem around the platform [18, p. 27].

Node.js applications can also benefit from certain framework modules that add *MVC* (Model-View-Controller) capabilities. One such module is the *express* framework[12], which includes features such as advanced routing and templates and brings *Node.js* one step closer to being a full-stack Web

[12] http://expressjs.com/

framework (see section 3).

Scalability

In the previous section, methods of implementing asynchronous behaviour in *Node.js* applications were listed. This section aims to explain the implications of executing this applications with regard to the execution environment.

Applications running on *Node.js* per default only use a single thread for processing [45]. As mentioned in the previous chapter, this has a positive effect on scheduling overhead. Because of the nature of *JavaScript* and *Node.js* (e.g. asynchronous networking and file abstractions, use of callbacks), it is comparably easy to write code that does not block the processing thread. However, *if* blocking occurs, the consequence is that the whole application is unable to process any requests until the blocking action has finished. On the other hand, this removes any need for synchronisation concerns and prevents address space conflicts between threads [7, p. 105].

To scale out a *Node.js*-based application, two main steps can be taken: Just scaling out on a single multi-core machine or scaling out on multiple machines. The first can be archived by creating multiple instances of the same program using the `cluster` module of *Node.js* (see program 3.5). This way, a master process has control over several child processes that handle requests asynchronously based on load balancing [18, p. 64]. The technique of having one process create child instances is called forking. If forking is not supported by the operating system (e.g. on Windows systems), the application creates multiple threads in the same process. Running the application on multiple servers has no special implications for *Node.js*; shared state has to be archived by a messaging protocol like *pub-sub* [18, p. 137].

Performance

Node.js is very suitable for massive connection concurrency and data-heavy applications [26, p. 44]. The *V8* engine executes *JavaScript* code at a very favourable speed; interfaces that often slow down browser-based applications (like the *DOM*(Document Object Model)) are not present in a server-side environment. However, the code is executed by interpretation, which is inherently slower than the execution of binary files or virtual machine bytecode (like *Java*). Figure 3.2 illustrates the serious impact of intensive computations on response time.

3.1.2 Eventmachine

Ruby[13] is a dynamic programming language that has a high adoption rate due to the popular *Ruby on Rails* MVC framework that powers a lot of

[13] https://www.ruby-lang.org

Program 3.5: The `cluster` module provides an abstraction of creating multiple instances of program execution. The first process running the code is defined as the master process and all other processes (the number of processes depends on the number of processing cores in the system) are forked as child processes. Code source: [18]

```
1 var cluster = require('cluster');
2 var http = require('http');
3 var numCPUs = require('os').cpus().length;
4
5 if (cluster.isMaster) {
6     // Fork workers.
7     for (var i = 0; i < numCPUs; i++) {
8         cluster.fork();
9     }
10    ...
11 } else {
12    // Worker processes have a http server.
13    http.Server(function(req, res) {
14        res.writeHead(200);
15        res.end("hello world\n");
16    }).listen(8000);
17 }
18
```

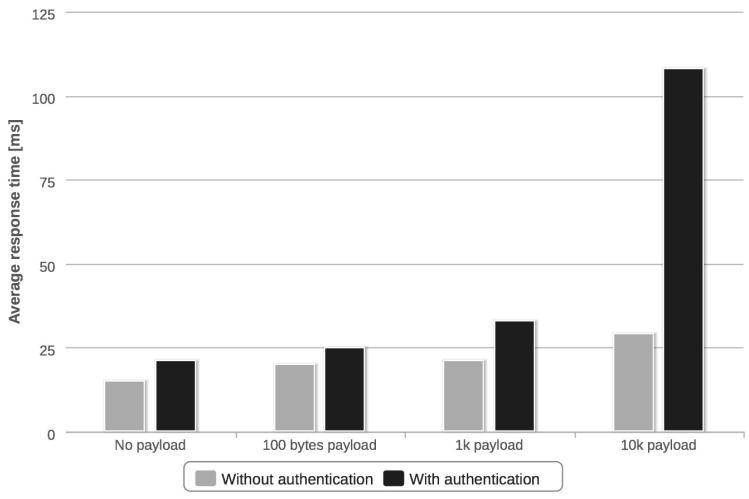

Figure 3.2: In this figure taken from a performance analysis of D. Torstensson and E. Eloff, requests are sent to a *Node.js* server with different payload sizes. The requests are sent both with and without authentication; authentication is done by hashing (i.e. processing) the whole payload using the *SHA1-HMAC* algorithm. Larger payloads are more computationally intensive and result in a longer response time. Image source: [26]

Program 3.6: This program demonstrates how blocks can be used with the `TCPServer` library (included in *Ruby's* standard library) to create a new thread for every incoming network client. The block (line 4 to 8) acts as a container applied to the result of previous operations, similar to a closure in *JavaScript*; the `client` variable is the result of the `new` method of the `Thread` class, which accepts a `TCPSocket` object. Code source: [35]

```
1 require 'socket'
2 server = TCPServer.new(2202)
3 while true
4     Thread new(server.accept){ |client|
5         msg = client.readline
6         client.write "You said: #{msg}"
7         client.close
8     }
9 end
10
```

modern Websites [22, p. 11]. However, unlike *JavaScript*, *Ruby* was not conceived with non-blocking event-driven behaviour in mind. *Eventmachine* is a library that aims to facilitate the process of developing non-blocking Web server applications in *Ruby*.

Development

As mentioned in section 3.1.1, *JavaScript* uses anonymous and first-order functions to manage asynchronous program flow. Due to *Ruby's* object-oriented nature, these concepts are not supported at language-level; instead, it supports so-called *blocks* that in some way can act like anonymous functions and receive parameters from previous operations [10]. See program 3.6 for a simple demonstration of how blocks can be used to create a basic networking server.

To create a non-blocking networking server in *Ruby*, more complex operations are needed. This includes creating and managing a complete event-loop[14] and using the `IO` class and the `accept_nonblock` method of the `TCPSocket` class to create logical concurrency on one thread (an exhaustive example can be seen in [35]). Operations like handling connections and reading input from the sockets also have to be managed by the developer [35].

EventMachine provides a simple way to abstract the process of managing event-based concurrency with *Ruby*. It includes its own event-loop – or *reactor* – which creates and handles events across the application. To interact with the environment the reactor provides asynchronous interfaces – called

[14]Strictly speaking, since no events are involved this is called a *reactor loop* [35].

Program 3.7: A simple echo server, i.e. a server that responds in a simple way depending on what the request contains. A *Ruby* module (line 5) contains the necessary logic and is managed by the *EventMachine* system. Line 10 initialises the reactor loop and line 11 starts the server using the predefined module.

```
1 require 'rubygems'
2 require 'eventmachine'
3
4 module EchoServer
5     def receive_data data
6         send_data "You said: #{data}"
7     end
8 end
9
10 EventMachine::run {
11     EventMachine::start_server "127.0.0.1", 2202, EchoServer
12 }
13
```

Connections, which have to be defined within the reactor loop. *EventMachine* includes several ways of creating connections:

- Creating a subclass of the `Connection` class and overriding its methods, then passing the class reference to the *connect* method of *EventMachine*
- Creating a module with the appropriate methods for handling connections (see program 3.7)
- Using a block (see program 3.6) and overriding methods of the connection object passed as parameter

Program 3.7 demonstrates, how the simple TCP server from program 3.6 can be implemented using the *EventMachine* reactor. Besides this comprehensible TCP communication functionality, *EventMachine* also includes functionality for deferring or postponing program logic. Deferring is important when interacting with code that would normally block the event-loop (see program 3.8). The `Timer` class or the `add_timer` and `add_periodic_timer` methods can be used to execute program logic at an arbitrary point in time (e.g. for scheduled or recurring tasks). There is also a `Queue` comprehension for managing multiple asynchronous tasks at once (cf. the `all` comprehension of the *Q* library in section 3.1.1).

Program 3.8: An example of using *EventMachine* to archieve *JavaScript*-like callback functionality in *Ruby*. A long-running operation can be put in a block, the execution of which is managed by *EventMachine* via its threadpool. After the execution has completed, the result is passed to another block (i.e. the "callback") as a parameter.

```
1 operation = proc {
2     # long-running operation, e.g. database query
3 }
4 callback = proc { |result|
5     # do something with result
6 }
7
8 EventMachine.defer(operation, callback)
9
```

Scalability and Performance

Like *JavaScript*, *Ruby* is an interpreted scripting language and as such performance is inherently inferior to compiled languages[15] (see section 3.1.1, *Performance*). When using the default *Ruby* VM[16], a security measure called *Global Interpreter Lock* prevents program threads from archiving physical concurrency by only executing one logical thread at one(see figure 3.3). This is done to prevent sharing non thread-safe code with other threads [34]. Thus, to scale a *Ruby* application running on the default virtual machine, several process instances have to be created. This is similar to *Node.js* and many implications that apply to scaling *Node.js* applications also apply to *Ruby*. *JRuby*[17] is an alternative implementation of the *Ruby* interpreter which theoretically allows for controlling physical concurrency at application level [34].

3.1.3 Others

There are numerous other examples of event-driven concurrency frameworks that are less documented or fitting to be presented in depth here. *React*[18] is a framework written in *PHP*[19], a scripting language that is often used in simple Web server applications [7, p. 36]. *Twisted*[20] is a reactor library for the *Python*[21] scripting language; its capabilities are similar to the *EventMa-*

[15] However, Web servers that focus heavily on I/O-bound operations like network and database communication may not need as much CPU performance as e.g. a server used for image processing.

[16] Virtual Machine, a program that executes code inside a dedicated environment.

[17] http://jruby.org/

[18] http://reactphp.org/

[19] https://php.net/

[20] https://twistedmatrix.com/

[21] https://www.python.org/

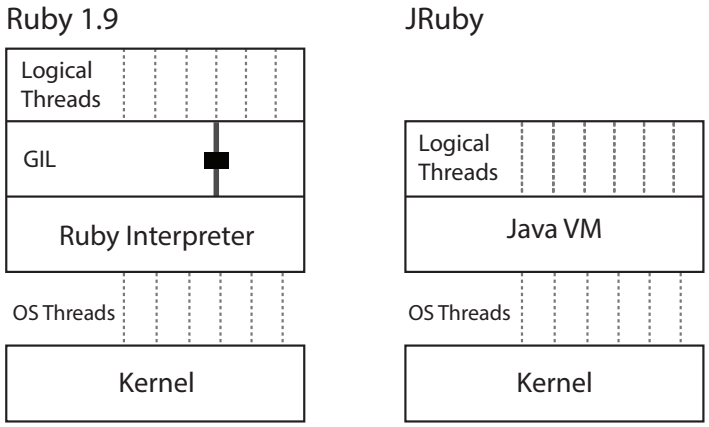

Figure 3.3: The Global Interpreter Lock of the *Ruby* VM prevents application logic to control parallel program execution on OS threads because it executes only one *Ruby* thread at once. However, with *JRuby*, this is possible. Image source: [34]

chine library presented in section 3.1.2. *Java NIO*[22] (see section 2.2.2) is a general interface for non-blocking I/O operations that allows for creating asynchronous Web server applications.

3.2 Actor-based technologies

3.2.1 Play!

Play![23] is a full-stack Web application framework for development in *Scala*[24] and *Java*. Since both languages are compiled to *Java* bytecode and run on the *Java* VM, both languages can be used side by side and libraries from the exhaustive *Java* ecosystem can be included; as of version 2, the *Play!* framework is written solely in *Scala* [16]. *Play!* uses the *Akka*[25] actor system, which is – like the *Scala* language and *Play!* itself – managed by *Typesafe Inc.*[26]; all three products are available as an integrated environment called *Activator*.

[22] Native Input and Output
[23] http://www.playframework.com/
[24] http://www.scala-lang.org/
[25] http://akka.io/
[26] https://typesafe.com/

Program 3.9: This program contains a very simple demonstration of how application-level code integrates with the *Play!* framework. `def` defines a new method, which is wrapped by the `Action` constructor method. The actual method logic is passed to the `Action` wrapper as a block, which has to return a `Result` object. The `Ok` method in line 4 converts a string to a `Result` object with the HTTP status code 200, indicating a successful operation with a non-empty response.

```
1 // Synchronous action
2 def shortProcessingRequest = Action {
3     val result = (2 + 2).toString
4     Ok(result)
5 }
6
```

Development

Play! is different from all aforementioned frameworks (see section 3.1), not only in the sense that it uses actor-based concurrency, but also in that it is based on a compiled programming language rather than an interpreted scripting language. *Scala* is a rather young[27] programming language that is syntactically very similar to *Java*, but extends it with functional programming paradigms and a more sophisticated type system. Since it has native support for comprehensions associated with concurrency and a more concise syntax, it is better suited for applications with a high amount of application-level concurrency operations [16, p. 9]; therefore, in this section *Play!*'s functionality is presented using *Scala*.

Play! builds upon the MVC[28] model, which means that incoming requests are handled by a user-defined controller structure. A controller contains *Actions*, i.e. methods that are called when a certain type of request arrives. Each controller method must return an object of the class `Result`, which contains the data to be sent back to the client [23, p. 27]. When defining an action, there are two basic types of actions in terms of concurrency – synchronous and asynchronous. Asynchronous actions must return an object of the type `Future[Result]`. A *Future* is similar to a *Promise* (see section 3.1.1, *Development*) and indicates that the result is not available at the momentary point of execution, but at an arbitrary time in the future; only when the calculation of the result has finished (either successfully or due to failure) the response is sent to the client (see program 3.9 and 3.10, respectively) [16, p. 86].

Scala provides various language features and libraries for handling concurrency like the `scala.concurrent` library, which includes versatile com-

[27] Introduced in 2003
[28] Model-View-Controller

Program 3.10: Returning asynchronous results is slightly more complex than returning synchronous results (see program 3.9). Using the `async` method of the `Action` object, a block returning a future result can be invoked. The `ContactDatabase.findById` method is a fictional database query that returns e.g. a `Future[Contact]` object. Since the block expects a `Future[Result]` object as a return value, the database result has to be *mapped* to an action result. The `map` method invokes a new block, which is executed when the future operation is resolved successfully. This block receives the non-future result as a parameter, which is wrapped by the `Ok` method. Thus, the result type of the statement in line 3 changes from `Future[Contact]` to `Future[Result]`.

```
1 // Asynchronous action
2 def longDatabaseRequest = Action.async {
3     ContactDatabase.findById(123) map {
4         result =>
5             Ok(result.toString)
6     }
7 }
8
```

prehensions for resolving one or multiple future results. However, *Play!* includes the *Akka* actor system and the respective libraries to even further facilitate concurrent processing. *Akka* is used by *Play!* internally for various tasks like request handling, but it can also be used at application-level [16, p. 83]. The actor system can be used for scheduling one-time and recurring operations, but its eponymous use is to manage actors (see section 2.2.4). In *Play!*, explicitly used actors are well-suited for autonomous tasks like handling communication with third-party Web services or sending emails. See program 3.12 for a code example and section 4 for more detailed use-case explanations.

Scalability

Scala – even though its name being a portmanteau of the words *scalable* and *language* – does not support scalable actor systems at language level. It only assumes the underlying host's thread model [13, p. 3]. The greater part of *Play!*'s scalability is achieved by the *Akka* actor system [11, p. 16]. How exactly this system behaves depends on the runtime configuration, which is defined via configuration files. Actors are generally associated with a certain *execution context*, i.e. a certain configuration of the actor system. In *Akka*, there are two main types of execution contexts or *executors*:

Program 3.11: This program is a demonstration of a simple actor used to send emails. The actor class inherits from the `Actor` class of the *Akka* library. As mentioned in section 2.2.4, actors do not share any state and communicate via messages. The actor class has to implement the `receive` method, which is called when messages arrive. Messages can have any type, but are usually sent via different *case classes* (see line 1 to 9), depending on the context of the message. For sending an email, these classes would contain for instance an email address and some text content. The `receive` method uses a block with *pattern matching* to determine the message type. Based on the message type, different actions can be taken by the actor. To send a message to an actor, the actor reference can be generated by the actor system (see line 28). Messages can be sent using the `!` method, the `?` method can be used to "ask" the actor, i.e. send a message and act upon a future response.

```
1  case class DefaultMail(
2      email: String,
3      content: String
4  )
5
6  case class ImageMail(
7      email: String,
8      image: String
9  )
10
11 class Mailer extends Actor {
12
13     def receive = {
14         case DefaultMail(email, content) =>
15             sendDefaultMail(email, content)
16         case ImageMail(email, image) =>
17             sendImageMail(email, image)
18     }
19
20     def sendDefaultMail ...
21
22     def sendImageMail ...
23
24 }
25
26 class Test {
27
28     val mailer = Akka.system.actorOf(Props[Mailer])
29
30     def test() = {
31         mailer ! DefaultMail("john@doe.com", "Hello John")
32     }
33
34 }
35
```

Thread pool executor Multiple worker threads are preallocated and incoming messages are distributed among free threads – this minimises thread overhead

Fork join executor If the amount of work for single messages exceeds a certain size, the task can be split among multiple processing cores by creating (i.e. *forking*) multiple instances of tasks that distribute work among them

Each execution context allows for configuring the minimum and maximum number of threads that are used as well as a multiplication factor that is based on the available cores. This allows for a very specific configuration of the actor system: If an execution context tends to dispatch many small tasks, the maximum number of threads can be increased, if there are few large tasks, fewer threads should be used [11, p. 105].

There is also a number of different dispatcher types to choose from, depending on whether actors should share a mailbox and the order in which actors are handled. Furthermore, the behaviour of mailboxes upon exhaustion can also vary from neglecting new messages to not being sent new messages [11, p. 104].

All this configuration options account to the single-system scalability of *Akka*. However, actors are not bound to reside on one single system. The means of communication between actors is not specified and can also be done using networking with remote systems. The default implementation of communication between different *Akka* systems uses TCP and the `akka://` URL scheme. The orchestration of the entirety of actor subsystems is done by a master node system that handles message dispatching [11, p. 233]. This way, *Akka* can be scaled out to a large number of systems.

Performance

Play! internally uses the *Netty*[29] HTTP server, which builds upon *Java NIO* (see section 3.1.3) to achieve non-blocking I/O capabilities [16, p. 52]. While *Netty* has proven to be capable of serving more than 40000 requests per second[30], this on only pure network communication without Web-specific processing and I/O involved. The same request-based tests conducted on a *Play!* application yield 9000 requests per second. Even with website-typical database operations and processing, a single *Play!* application can serve 1350 to 2400 request per second [40]. What is also noteworthy is that due to the adaptive nature of its actor system, *Play!* delivers formidable performance even without the need for any application-level configuration.

[29] http://netty.io/

[30] Tested on an *Amazon EC2* cluster (http://aws.amazon.com/ec2/)

3.2.2 Lattice

Lattice[31] is a lightweight Web framework for the *Ruby* scripting language (see section 3.1.2). It actually represents a combination of several different technologies that make up the entirety of the framework. *Lattice* uses the *Celluloid* actor system[32] for processing and builds upon the *Reel*[33] Web server. On application level, it uses the *Ruby* port of *Webmachine*[34] (originally written in *Erlang*[35]) to facilitate the handling of HTTP requests by mapping URL routes to respective controller methods.

Celluloid actors can be created by simple including the `Celluloid` object inside a *Ruby* class; see program 3.12 for a detailed example [44]. When using the `async` method on an actor (as seen in program 3.12), no value is returned. However, using the `future` method returns a `Celluloid::Future` object, which represents a rather primitive promise (see section 3.1.1). There is also an explicit way of creating promises by creating a new instance of the `Celluloid::Future` object with a block parameter containing the asynchronous logic. The only way to resolve a promise is to block current execution and wait for completion; there are no comprehensions like mapping or resolving multiple promises at once [29].

3.2.3 Others

Apart from the above frameworks, there are hardly any full-stack Web frameworks that use the actor concurrency model. *Lift*[36] is another example of a *Scala*-based actor-driven framework that is similar to *Play!*. However, there is no option to develop applications in *Java*, it does not offer versatile actor functionality compared to *Akka* and the ecosystem (i.e. support by other developers) is not as advanced as with *Play!*. *Xitrum*[37] is another *Scala*- and *Akka*-based Web framework that aims to offer functionality similar to *Lift*. *spray*[38] is a lightweight general-purpose I/O framework also based on *Scala* and *Akka*. It offers a number of modules for extension, including *spray-can* and *spray-http*, which can be used to implement a low-level HTTP server.

[31] https://github.com/celluloid/lattice
[32] http://celluloid.io/
[33] https://github.com/celluloid/reel
[34] https://github.com/seancribbs/webmachine-ruby
[35] http://www.erlang.org/
[36] http://liftweb.net/
[37] http://xitrum-framework.github.io/
[38] http://spray.io

Program 3.12: This program is an adaption of the actor example presented in program and demonstrates how to create actors with *Celluloid*. By including the `Celluloid` object in a default *Ruby* class body, actor functionality is added to the class (line 4). When creating an instance of this class, a new *Celluloid* actor is initialised (line 18). To execute an asynchronous routine, the `async` method of the actor object has to be called, followed by the respecitve method name (line 22).

```
1
2 class Mailer
3
4     include Celluloid
5
6     def send_default_mail(email, content)
7         send_mail(...)
8     end
9
10    def send_image_mail(email, image)
11      send_mail(...)
12    end
13
14 end
15
16 class Test
17
18    mailer = Mailer.new
19
20    def test()
21
22        mailer.async.send_default_mail("john@doe.com, "Hello John")
23
24    end
25
26 end
27
```

Chapter 4

Implementation

This chapter is a documentation of experiences gained during the creation of a concurrent high-performance Web application[1] – from deciding on the programming language, the Web framework as well as supplementary technologies to developing, deploying and maintaining the actual application. Technical aspects (see chapter 2) as well as the current state of the art (see) play a dominant role during the whole process; however, this chapter is not limited to applying previous content in practice, but rather uses it as a base for gaining contextual and more detailed knowledge about the topic.

4.1 Prerequisites

4.1.1 Requirements

Depending on the projected success and behaviour of the application, the following requirements were decided on by the company:

- The application is a social network. This implies a high request frequency and many atomic database operations.
- There should be client applications for Web and mobile operating systems, so the communication between applications should be flexible.
- Several third-party Web services have to be included, either via pure HTTP interfaces or using native libraries.
- Since the success of the application can not be predicted, a high scaling range – also using multiple servers – is necessary.

Apart from these enterprise decisions, another requirements was added with respect to an interesting and innovative nature of the project: The application should make heavy use of event- or actor-driven programming paradigms.

[1] At the time of writing, this Web application is live and already has several thousand users via Web and mobile clients. Unfortunately, due to corporate secrecy, the name of the application can not be disclosed.

4.1.2 Language and Framework

The first and probably most important step in creating an application is to decide on a programming language and framework[2]. Research about modern event- or actor-driven frameworks yielded several possibilities, most of which are listed in chapter 3. Based on the extent of framework documentation, interoperability with other technologies and community size, the two final options were *Node.js* and *Play!*.

While *Node.js* has the advantage of supporting a simple and widespread programming language that includes concurrency by design, *Play!* appealed by supporting a type-safe, object-oriented language (either *Java* or *Scala*) with a solid set of libraries due to *Java*'s history as a popular enterprise language. Furthermore, scaling on single systems is handled very differently by the two frameworks with *Node.js* leveraging multiple process instances and *Play!* using an actor system. Finally, *Play!* was chosen due to the better application structure and the superior CPU utilisation (considered that there may be some minor image processing operations).

This leaves two choices of language: *Java* and *Scala*. Even though *Scala* is not as widespread as *Java* – which may result in difficulties finding developers – it offers various advanced features including syntax simplifications and library-level functionality for concurrent operations, option data types and number ranges as well as a purely object-oriented structure[3].

4.1.3 Drivers and Libraries

The present Web application features two kinds of data storage: A *MongoDB*[4] database is used for persisting any long-lived information and a *Redis*[5] key-value store is used for short-lived information like caching as well as for pub-sub communication (details follow in section 4.2). Both components are accessed as *SaaS*[6] due to simple deployment and maintenance. A number of different drivers expose libraries to facilitate communication with these technologies; unfortunately, currently only few drivers that support asynchronous non-blocking I/O. Using blocking data access drivers with *Play!* would eliminate most of the performance gains achieved by *Play!*'s non-blocking I/O due to the occupation of processing threads.

The only asynchronous *MongoDB* driver at the time of writing was *ReactiveMongo*[7], a driver implementation written in *Scala* that basically exposes

[2] Since these two elements depend on one another, deciding on a framework inherently limits the choice of languages.

[3] *Java* has an inconsistent type system with types like `int` or `boolean` not being part of the global object hierarchy.

[4] http://www.mongodb.org/

[5] http://redis.io/

[6] Software as a Service, third-party companies that offer provision and maintenance of software on their own servers.

[7] http://reactivemongo.org/

the *MongoDB* API[8] to the application without any additional features like included *DAO*[9] functionality. However, this enables a very flexible way of interacting with the database, which is especially suitable for atomic operations like increasing a single numeric value or deleting a property. *ReactiveMongo* offers a *Play!* plugin for easy integration with the framework (e.g. by managing connections according to application start/stop).

For interfacing with the *Redis* server, the *rediscala*[10] driver proved to be a good choice by offering non-blocking access to the most important server operations. *rediscala* even offers dedicated actor superclasses designed for use with *Akka* (see section 4.2.4). On the downside, *rediscala* does not provide a dedicated *Play!* plugin, thus custom framework integration had to be implemented in order to use the driver.

The application also makes use of several other libraries, e.g. for sending emails. Here, a great advantage of *Scala* comes into play: Due to being compiled to *Java* bytecode, *Scala* is binary compatible with all available *Java* libraries. For instance, the *Apache Commons*[11] email implementation written in *Java* can also be used to send emails in *Scala*.

4.2 Development

4.2.1 Requests and Actions

As already mentioned in section 3.2.1, *Play!* uses different controller actions to determine if a request should be served synchronously or asynchronously (see program 3.9 and 3.10, respectively). A good example for a synchronous request is an action that returns the current server time for the request signing procedure[12]. Here, no database action is necessary and the retrieval of system time does not consume much processing time. However, nearly all requests to the Web server involve some kind of database operation; either resources are read or written or a combination of multiple operations is executed. When writing a value to the database, the response is served after the operation completes to indicate success or failure to the client; this way, the client can decide for itself whether it waits for the response or, for instance, updates the user interface right after sending the request.

[8] Application Programming Interface

[9] Data Access Object, a common feature in database drivers to simplify storage and retrieval of code objects in the database.

[10] https://github.com/etaty/rediscala

[11] http://commons.apache.org/

[12] Requests are only valid for a certain timespan to prevent *replay attacks*, i.e. capturing and sending a request multiple times.

Program 4.1: In this example, two images are uploaded to a remote server. Only when both uploads have completed, the response should be sent containing the URLs oft both images. The `for` comprehension receives a block with multiple `Future[String]` assignments. The `yield` statement wraps these `Future` objects in a single `Future[(String, String)]` object. This is a type called a *tuple*, i.e. two objects combined into one. The `map` comprehension in line 4 maps this `Future` to a simple tuple, the values of which can be retrieved using the `._1` and `_.2` properties (line 6).

```
1 (for {
2     picture1Url <- uploadPicture(picture1)
3     picture2Url <- uploadPicture(picture2)
4 } yield (picture1Url, picture2Url)) map {
5     result =>
6         Ok("Here are your pictures:\n" + result._1 + "\n" + result._2)
7 }
```

4.2.2 Basic Asynchronous Operations

Working With Futures

The majority of asynchronous operations involve database or cache access. The database driver and the cache driver both return `Future` objects, i.e. the respective calls return almost instantaneously and yield a value that is resolved later (cf. program 3.10). In the simplest case, this value can be mapped to a `Result` object and returned by an asynchronous action. However, this is not always the case; frequently, the returned value has to be processed and results are even used as parameters for new database operations. Program 4.2 shows an example with two nested `Future` resolutions.

To resolve multiple `Future` objects in parallel and work with the combined results of the single asynchronous operations, the `for` comprehension can be used; see program 4.1 for an example.

Apart from database and cache operations, `Future` objects also result from using *Play!*'s integrated `WS` Web service library. Since HTTP requests take an arbitrary amount of time to return, the use of asynchronous processing yields high performance gains since this way, a potentially slow third-party Web server only delays the application's response to the client, but does not inflict the application's performance by blocking threads.

Deferring Program Flow

Certain operations are not relevant to the further program flow and can be executed concurrently without the need for resolving return values. These *asynchronous side-effects* can be executed at any point during program flow. For instance, if the user requests that his photo album should be deleted,

Program 4.2: This is an basic example of how two database operations can be nested in a *Play!* application. A typical occurrence of two database operations is for instance when a new user should be created with a unique username. The outmost block is the default asynchronous `Action` block with a *body parser* as argument (line 1). This body parser converts the text from the request body into a JSON (JavaScript Object Notation, commonly used for HTTP communication) object suitable for further processing. This body must contain a desired username, which is obtained by traversing the JSON abstract syntax tree (using the \ method). Next, a database query is initiated using the provided username. This query returns a `Future[Option[User]]` object; the `Option` type indicates that the value can either be present (`Some`) or absent (`None`). The `flatMap` method is similar to the `map` method, but instead of `Result` objects, all statements inside the block must return `Future[Result]` objects. If the database query returns an object of the type `None` (line 5, 7, 9 and 12 are examples of *pattern matching*), no user with the given username is found and thus the new user can be inserted and the result of the database operation can be mapped to a `Result` using `map`. However, if the username already exists, no subsequent database operation has to be initiated and the response can be sent instantly. To generate a readily resolved `Future`, the `Future.successful` method can be used. `Created`, `Conflict` and `InternalServerError` are helpers for the response status codes `201`, `409` and `500`, respectively.

```
1 def insertUniqueUser() = Action.async(parse.json) {
2     request =>
3         val username = (request.body \ "username").as[String]
4         UserService.findByUsername(username) flatMap {
5             case None =>
6                 UserService.insert(request.body) map {
7                     case Some(id) =>
8                         Created("New user created with id " + id)
9                     case None =>
10                        InternalServerError("User could not be created")
11                }
12             case Some(user) =>
13                Future.successful(Conflict("Username exists!"))
14        }
15 }
16
```

the request may return as soon as the album object is removed from the database, but the deletion of the actual image files (which may take some time) can be deferred to a later point in time:

```
1 def deleteAlbum(id: String) = Action.async {
2     AlbumService.deleteById(id) map {
3         case true =>
4             ImageService.deleteForAlbum(id)
5             Ok("Your album was deleted!")
```

```
6         case false =>
7             InternalServerError("Something went wrong!")
8     }
9 }
```

Deferring execution can also be done using *Akka*'s scheduling functionality. The present application uses this scheduling functionality to obtain a new *access token* for authentication from Web services, depending on when the old token expires. The execution can be scheduled at a specific point in time or repeated periodically:

```
1 import play.api.libs.concurrent.Akka
2
3 Akka.system.scheduleOnce(10.minutes)(sendReminderEmail())
4
5 // The first parameter defines the initial delay, the second one the interval
6 Akka.system.schedule(Duration.Zero, 30.minutes)(renewAccessToken())
```

Technically, actors can also be used to defer program flow, but are generally used for more sophisticated operations (see section 4.2.3).

Converting Blocking Code

Of course, not all operations return asynchronous results, especially when using third-party or *Java* libraries. Wrapping blocking method calls in `Future` objects that can be resolved by *Akka* is rather trivial:

```
1 def asynchronousOperation(param: String): Future[String] = {
2     Future {
3         synchronousOperation(param)
4     }
5 }
```

When resolving `Future` objects within a *Play!* application, *Play!*'s default dispatcher is used (for information about dispatchers see section 3.2.1, *Scalability*). However, especially for computationally expensive operations like image processing it is advisory to use a dedicated dispatcher. New dispatchers can be created by defining them in the *Akka* configuration within *Play!*'s configuration files:

```
1 akka {
2     actor {
3         image-processing-dispatcher {
4             fork-join-executor {
5                 parallelism-max = 2
6             }
7         }
8     }
9 }
```

This defines a dispatcher called `image-processing-dispatcher` that uses a *fork-join executor* and at most two threads in order not to block the application. Program 4.3 gives an example of an expensive image processing

Program 4.3: This program shows how a comparably expensive image processing operation can be deferred using a custom dispatcher. In line 1, a reference to the custom dispatcher is created using *Akka*'s lookup functionality. The `Future` block (line 5 to 19) wraps the expensive operation and defines the dispatcher that should be used to resolve the `Future` object (i.e. how it should be processed by the actor system). The operation itself (line 15) consists of a operating system call to the *ImageMagick* (http://www.imagemagick.org/) command line tool. The `waitFor` method blocks the dedicated dispatcher thread until the processing has finished. After that, the `Future` object is resolved with a `File` reference to the generated image.

```
1 val dispatcher = Akka.system.dispatchers.lookup("akka.actor.image-
      processing-dispatcher")
2
3 def generateImage(filename: String): Future[File] = {
4
5     Future {
6
7         val cmd = Array(
8             "convert",
9             "-background", "black",
10            "-fill", "white",
11            ...
12            filename
13        )
14
15        Runtime.getRuntime.exec(cmd).waitFor()
16
17        new File(filename)
18
19    } (dispatcher)
20
21 }
22
```

operation converted to an asynchronous operation that can be deferred using a custom dispatcher.

4.2.3 Actor-based Operations

Since *Play!* does not expose the underlying actor structure to its libraries, application can be built without explicitly using any actors. However, the present application uses four actors for complex concurrency operations: The `Mailman` and `Notifier` actors are used to defer and isolate complex asynchronous operations, namely sending emails and mobile notifications, respectively. These two actors are structurally rather similar to the `Mailer` actor presented in program 3.12, but include extensive logic to generate the different notifications depending on the received actor message. However, the

`Feeder` and `Subscriber` actors are more complex and play an important role in section 4.2.4.

4.2.4 Advanced Actor Usage

The application also includes a news feed using a technology called *Websockets* to send events to subscribed client applications over TCP. The basic idea is that when one user takes a specific action, other users that are interested in that action get notified instantly. A rather trivial approach would be to keep a collection of currently subscribed clients and notify them directly when a certain event occurs according to what events they have subscribed to. However, as defined in section 4.1, *Requirements*, the application should be scalable to multiple systems. This introduces a problem: Clients that subscribed on one particular system will not get events that occurred on other systems since the event is not propagated across all systems.

The solution is to use a centralised messaging system. Options include dedicated protocols like *AMQP*[13], but since the application already uses *Redis*, which supports the *Pub/Sub*[14] paradigm, using this system is more practicable. *Pub/Sub* works by publishing messages to the central server, which forwards them to all systems who have subscribed to the corresponding message type.

Publishing to the *Redis* server can be easily done using the *rediscala* library; for instance, the following line of code is used to publish a message when a user `abcd` likes a photo `1234`:

```
1 RedisService.publish("/picture/1234/likes", "abcd")
```

Due to the side-effect nature of publishing a message, this may also be done using an actor.

Subscribing and receiving messages is the more complex part of the centralised message passing lifecycle. Ideally, incoming *Redis* messages should be translated to *Akka* messages for subsequent handling inside the actor system. Fortunately, *rediscala* includes the `RedisSubscriberActor` superclass to facilitate subscribing to messages. Program 4.4 shows how this subscriber actor class is structured and used.

The fourth and last actor, is the `Feeder` actor. This is a standard actor that listens for two types of messages:

- If a client connects to the application via *Play!*'s `WebSocket.tryAccept` controller action, the actor receives a message containing the desired feed (e.g. `/picture/1234/likes`) and a reference handle to the client. The actor then stores the client inside a collection.
- If a feed message arrives, the actor iterates over its client collection and identifies clients that have indicated interest in the message. The

[13] Asynchronous Message Queuing Protocol
[14] Publish-Subscribe

Program 4.4: This program shows, how the `RedisSubscriberActor` superclass can be used to create an actor class that listens for custom *Redis* `publish` messages. Besides the publish channel, *Redis* messages may include a pattern signature; messages are then only distributed to the subscibers that signify interest in the particular pattern. To be able to use the `RedisSubscriberActor` superclass, the subscriber actor has to supply two methods. The `onMessage` method is called when a message without a pattern arrives; however, because in this case only pattern messages are used, this message can return an empty object. The `onPMessage` method receives pattern messages and *tells* the `Feeder` actor to send the message to the currently connected clients. The subscriber actor has to be initialised upon application start; this can be done using *Play!*'s `Global` object, which can override the `onStart` method (line 15). Inside the `onStart` method, the subscriber actor is created using the `Props` object, which creates a new class instance using specified constructor parameters. The second parameter, `Nil`, indicates, that the subscriber should not listen to a particular channel; the third parameter is a sequence of patterns consisting of one pattern that matches likes for any picture. Note that this actor uses *rediscala*'s own dispatcher.

```
1 class Subscriber(channels: Seq[String] = Nil, patterns: Seq[String] =
      Nil) extends RedisSubscriberActor(Redis.socket, channels, patterns,
      Redis.password) {
2
3     def onMessage(message: Message) {
4         Nil
5     }
6
7     def onPMessage(message: PMessage) {
8         Feeder.push((message.channel, message.data))
9     }
10
11 }
12
13 object Global {
14
15     override def onStart(app: Application) = {
16         Akka.system.actorOf(Props(classOf[Subscriber], Nil, Seq("/
      picture/*/likes")).withDispatcher("rediscala.rediscala-client-worker
      -dispatcher"))
17     }
18
19 }
20
```

actor then sends relevant information (like which user has liked which picture) to the client over the *WebSocket*.

4.3 Deployment and Scaling

The described *Play!* application can run on any platform that can execute *Java* bytecode. Since it even includes its own Web server, it represents an integrated container that is readily suitable for deployment over multiple server instances. The present application is designed to use identical replications of actor systems on every system, the only means of sharing application state being the message passing via the *Redis* server (see program 4.4). This means that the application can theoretically be scaled out indefinitely, provided a *load balancer* serves incoming requests fast enough to the server instances and the database and cache communication happens at a formidable speed.

A *Play!* application could be scaled out in a different way: Instead of automated replications, the application could be designed using different modules on different systems that communicate via a central *Akka* system. This way, a number of systems could handle network I/O and other systems could handle side effects or intensive computations like image processing.

Chapter 5

Evaluation

In the previous chapter, the implementation characteristics of an actor-based Web framework were observed from the view of a developer (see chapter 4). While these aspects are important during the development of an application, once the application is publicly accessible, performance is a paramount factor. This chapter documents tests conducted with respect to performance criteria defined in chapter 2 – for example request frequency and response time (see section 2.1.4) – and aims to define use-cases for synchronous and asynchronous application structure in the context of Web frameworks.

5.1 Prerequisites

Modern Web server performance can be defined by the system's behaviour when processing a high number of simultaneous requests. Ideally, the response time for each request should be as low as possible and should not increase significantly with the number of simultaneous requests.

To test the behaviour of thread-based applications side by side to event-based applications, tests should preferably be conducted under very similar conditions, the only major difference being asynchronous processing. Since in section the *Play!* framework was identified as a feasible choice for demonstrating asynchronous processing in Web frameworks, *Play!* is also used in the following performance tests. To achieve comparable performance, the thread-based contender application should ideally also be run on the *Java* virtual machine (JVM). This leaves numerous choices; however, the *Spring MVC*[1] framework is considered a similar solution regarding application structure and runtime behaviour [16, p. 109].

[1] http://spring.io/

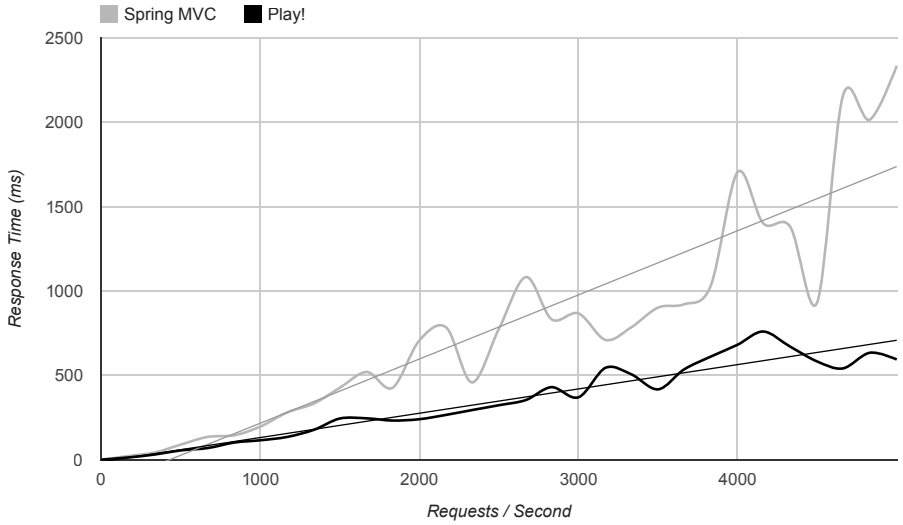

Figure 5.1: fib(1000)

5.2 Testing

Methodology

Independent, but identical systems

Netty vs Jetty Iterative Fib proved to be most suitable (tail recursion) Java VM 7 Not too much load, equal reliability

5.3 Results

JMeter loader.io

5.4 Interpretation

On-system operations!! => at maximum load, equal outcome from a certain point on

Off-system operations? DB? Web?

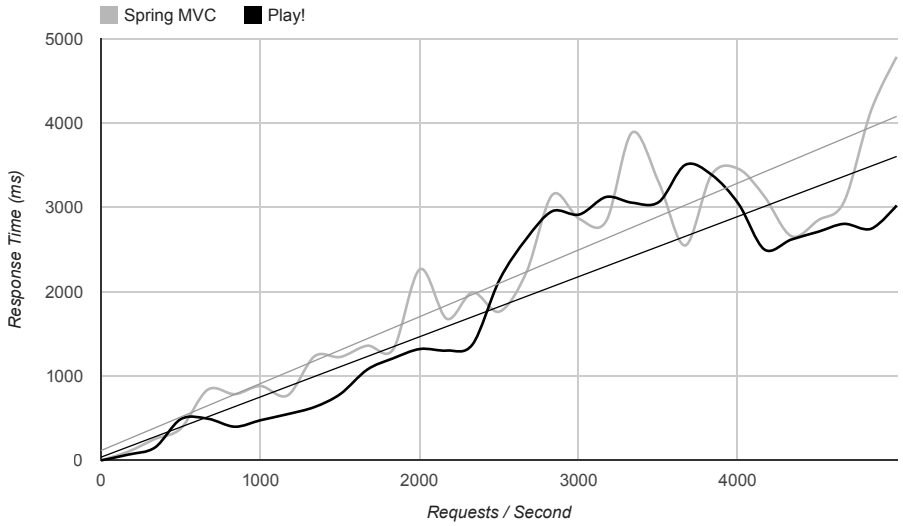

Figure 5.2: fib(5000)

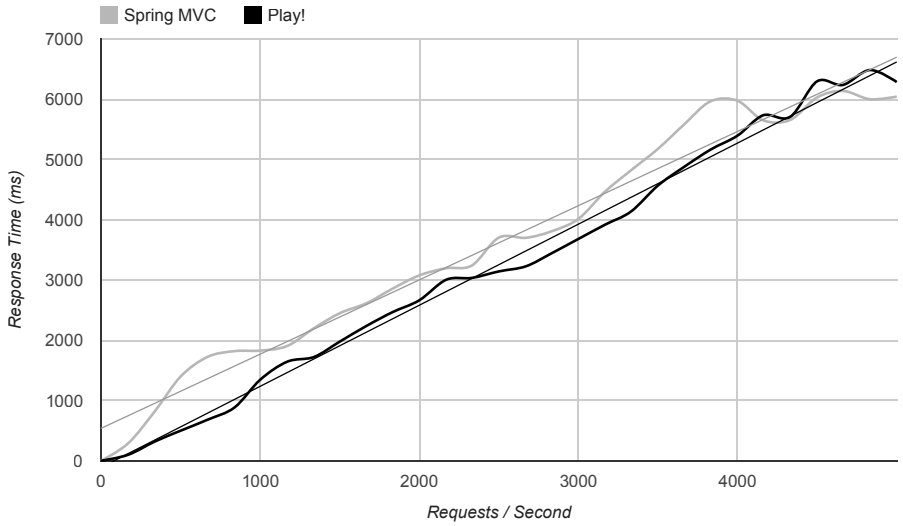

Figure 5.3: fib(10000)

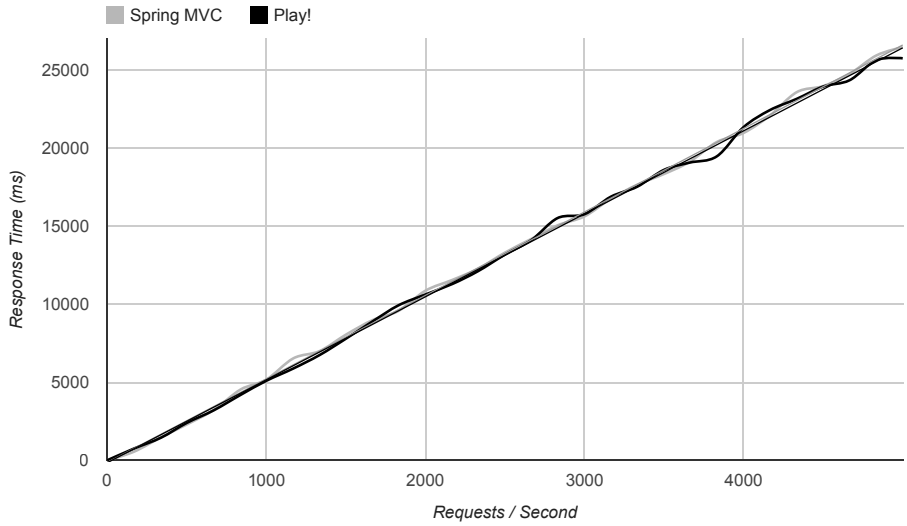

Figure 5.4: fib(50000)

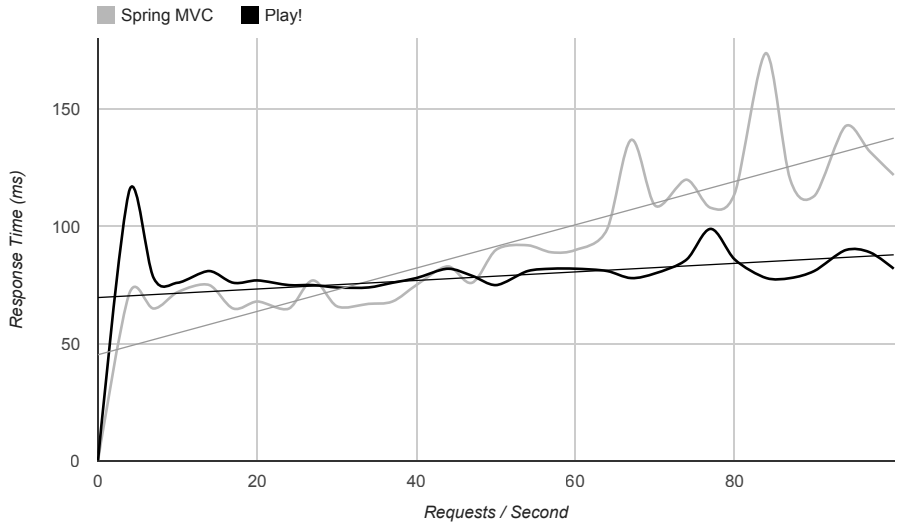

Figure 5.5: web(100)

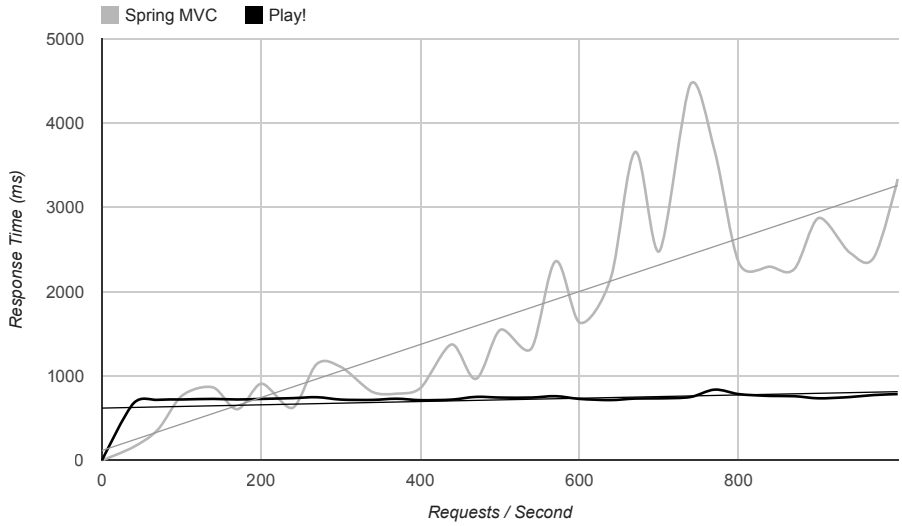

Figure 5.6: web(1000)

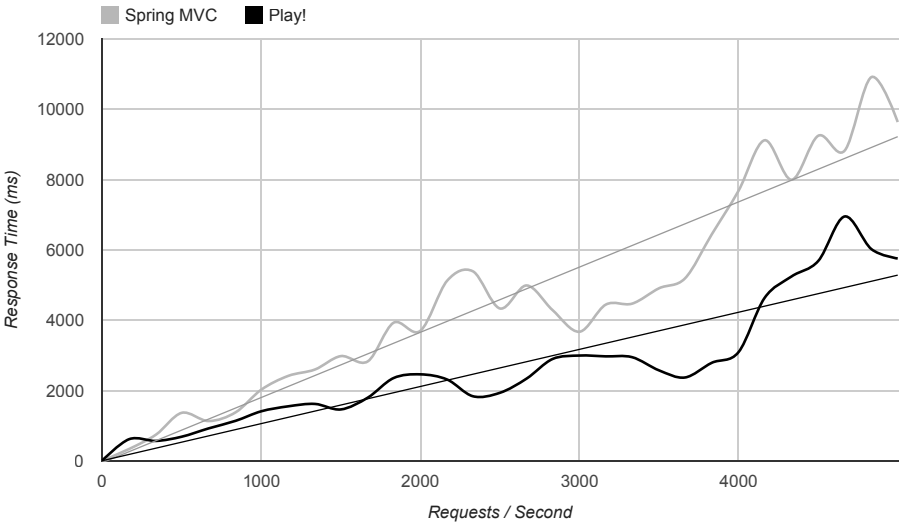

Figure 5.7: web(5000)

Chapter 6

Conclusion and Future Development

References

Literature

[1] J Armstrong. *Programming Erlang: SoÅ¿tware for a Concurrent World*. 2007. URL: http://dl.acm.org/citation.cfm?id=1403889.

[2] Rob Von Behren, Jeremy Condit, and Eric Brewer. "Why Events Are A Bad Idea (for high-concurrency servers)". In: *HotOS IX : The 9th Workshop on Hot Topics in Operating Systems* (2003).

[3] Jonas Bonér. "The Reactive Manifesto". 2013.

[4] Daniele Bonetta, Danilo Ansaloni, and Achille Peternier. "Node.Scala: Implicit Parallel Programming for High-Performance Web Services". In: *Lecture Notes in Computer Science Volume 7484* (2012).

[5] David Carrera et al. "Evaluating the Scalability of Java Event-Driven Web Servers". In: *ICCP 2004* (2004).

[6] Vaarnan Drolia, Cedric Ansley, and Chin Shen. "Threads vs Events for Server Architectures". 2010.

[7] Benjamin Erb. "Concurrent Programming for Scalable Web Architectures". PhD thesis. University Ulm, 2012.

[8] Joakim Eriksson. "Representation of asynchronous communication protocols in Scala and Akka". PhD thesis. Linköpings Universitet, 2013.

[9] Jeffrey Fischer, R Majumdar, and Todd Millstein. "Tasks: language support for event-driven programming". In: *PEPM '07 Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (2007), pp. 134–143.

[10] Michael Fitzgerald. *Learning Ruby*. O'Reilly Media, Inc, 2007. URL: http://books.google.com/books?hl=en\&lr=\&id=pYS_Fm5LqUYC\&oi=fnd\&pg=PR11\&dq=Learning+Ruby\&ots=YmKGFpwTaj\&sig=KQ5BeDl83fTDqqXppkoeumgtnyM.

[11] Munish K. Gupta. *Akka Essentials*. Packt Publishing, 2012.

[12] Philipp Haller and Martin Odersky. "Event-Based Programming without Inversion of Control". In: *Modular Programming Languages* (2006). URL: http://link.springer.com/chapter/10.1007/11860990_2.

[13] Philipp Haller and Martin Odersky. "Scala Actors: Unifying Thread-based and Event-based Programming". In: *Theoretical Computer Science* 410.2-3 (Feb. 2009), pp. 202–220. URL: http://linkinghub.elsevier.com/retrieve/pii/S0304397508006695.

[14] Philipp Haller and Frank Sommers. *Actors in Scala*. 1st ed. 2011. URL: http://scholar.google.com/scholar?hl=en\&btnG=Search\&q=intitle:Scala\#0 http://dl.acm.org/citation.cfm?id=2222540.

[15] Cal Henderson. *Building Scalable Web Sites*. O'Reilly Series May. O'Reilly Media, Inc, 2006, p. 330.

[16] Peter Hilton, Erik Bakker, and Francisco Canedo. *Play for Scala: Covers Play 2*. 1st ed. Manning Publications Co., 2013, p. 300.

[17] Gregor Hohpe. "Programming Without a Call Stack â Event-driven Architectures". In: *Enterprise Integration Patterns* (2006).

[18] Tom Hughes-Croucher and Mike Wilson. *Node - Up and Running*. 2012.

[19] Edward A. Lee. "The problem with threads". In: *Computer* 39.5 (May 2006), pp. 33–42.

[20] Paul Mackay. "Why Has the Actor Model Not Succeeded?" In: *surprise 97* (1997).

[21] S Nadimpalli and S Majumdar. "Techniques for Achieving High Performance Web Servers". In: *Parallel Processing, 2000. ...* (2000).

[22] R Orsini. *Rails Cookbook*. O'Reilly Media, Inc, 2008.

[23] Alexander Reelsen. *Play Framework Cookbook*. Packt Publishing, 2011.

[24] Sencha Inc. "Web Applications Come of Age". 2011.

[25] Stefan Tilkov and Steve Vinoski Verivue. "Node.js : Using JavaScript to Build High-Performance Network Programs". In: *IEEE The Functional Web* (2010).

[26] Daniel Torstensson and Erik Eloff. "An Investigation into the Applicability of Node. js as a Platform for Web Services". PhD thesis. 2012.

[27] Bryan Veal and Annie Foong. "Performance Scalability of a Multi-Core Web Server". In: *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems - ANCS '07* (2007), p. 57.

[28] Matt Welsh, David Culler, and Eric Brewer. "SEDA : An Architecture for Well-Conditioned, Scalable Internet Services". In: *SOSP-18* (2001).

Online Sources

[29] Tony Arcieri. *Celluloid Futures*. 2012. URL: https : / / github . com / celluloid/celluloid/wiki/Futures.

[30] Alexander Cogneau. *Node.js Events and EventEmitter*. 2013. URL: http://www.sitepoint.com/nodejs-events-and-eventemitter/.

[31] R Fielding, U C Irvine, and J Gettys. *Hypertext Transfer Protocol – HTTP/1.1*. 1999. URL: http://www.ietf.org/rfc/rfc2616.txt.

[32] Jesse James Garrett. *AJAX : A New Approach to Web Applications*. 2005. URL: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/.

[33] Brian Goetz. *Thread Pools and Work Queues*. 2002. URL: http://www.ibm.com/developerworks/java/library/j-jtp0730/index.html.

[34] Ilya Grigorik. *Parallelism is a Myth in Ruby Ruby*. 2008. URL: https://www.igvita.com/2008/11/13/concurrency-is-a-myth-in-ruby/.

[35] Aman Gupta. *EventMachine*. 2010. URL: https://dl.dropboxusercontent.com/u/635/em_export.pdf.

[36] Michael McCool. *The Serious Drawbacks of Explicit Multi- Threading*. 2008. URL: http://software.intel.com/en-us/blogs/2008/06/05/nitrogen-narcosis-part-ii-the-serious-drawbacks-of-explicit-multi-threading/.

[37] Mark McGranaghan. *Threaded vs Evented Servers*. 2010. URL: http://mmcgrana.github.io/2010/07/threaded-vs-evented-servers.html.

[38] Svetlin Nakov. *Internet Programming with Java*. 2004. URL: http://www.nakov.com/inetjava/lectures/part-1-sockets/InetJava-1.3-Multithreading.html.

[39] Oracle. *Establishing Performance Goals*. 2010. URL: http://docs.oracle.com/cd/E19900-01/819-4741/fygaj/index.html.

[40] Christian Papauschek. *Real-world performance of the Play framework on EC2*. 2013. URL: http://blog.papauschek.com/2013/04/real-world-performance-of-the-play-framework-on-ec2/.

[41] Michael Peterson. *Events and Event-Driven Architecture : Part 1*. 2012. URL: http://thornydev.blogspot.co.at/2012/01/events-and-event-driven-architecture.html.

[42] Pingdom. *A History of the Dynamic Web*. 2007. URL: http://royal.pingdom.com/2007/12/07/a-history-of-the-dynamic-web/.

[43] Mark Russinovich. *Pushing the Limits of Windows : Processes and Threads*. 2009. URL: http://blogs.technet.com/b/markrussinovich/archive/2009/07/08/3261309.aspx.

[44] Jesse Storimer. *Basic Celluloid Usage*. 2013. URL: https://github.com/celluloid/celluloid/wiki/Basic-usage.

[45] Mikito Takada. *Understanding the Node.js Event Loop*. 2011. URL: http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/.

[46] Webopedia. *Multi-core Technology*. 2007. URL: http://www.webopedia.com/TERM/M/multi_core_technology.html.

[47] Matt Welsh. *A Retrospective on SEDA*. 2010. URL: http://matt-welsh.blogspot.co.at/2010/07/retrospective-on-seda.html.