Using Event- and Actor-Driven Paradigms to Increase Web Application Performance

Felix Hessenberger

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 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, September 26, 2014

Felix Hessenberger

Contents

Abstract

Due to the ever-increasing popularity of websites and mobile applications, demands on Web servers continue to grow. Well-developed software can help to limit hardware cost while boosting performance. Event- and actor-driven development paradigms aim to depart from traditional modalities in order to better suit modern Web server applications. Various technologies already implement different approaches to these patterns. This thesis aims to elaborate upon how, how much and under which circumstances these technologies can increase Web server performance.

Kurzfassung

Die immer weiter steigende Popularität von Websites und mobilen Applikationen führt zu stetig wachsenden Anforderungen an Webserver. Gut durchdachte Software kann Hardwarekosten eindämmen und gleichzeitig die Leistung steigern. Event- und Actor-basierte Entwicklung weicht von bestehenden Lösungen ab, um bessere Ergebnisse für moderne Webserveranwendungen zu erzielen. Verschiedene Technologien implementieren bereits jetzt unterschiedliche Ansätze dieser Entwicklungsweisen. Diese Arbeit versucht herauszufinden, wie, in welchem Ausmaß und unter welchen Bedingungen diese Technologien die Leistung von Webservern steigern können.

Chapter 1

Introduction

No other recent technology has changed the way people interact as much as the World Wide Web. The World Wide Web—or Web, for short—offers a wide range of functionality, from simply displaying static content to instant messaging to extensive social networks. The popularity of the Web has spread from appealing solely to first-world businesses and younger people to almost all demographic sections and geographic areas. Instead of merely "browsing" the Web on an individual basis, today's Web is characterised by communication and collaborative consumption and creation of different forms of digital content. Devices for Web access are no longer limited to personal computers, but include a variety of categories such as smartphones and wearable computing and even more ubiquitous forms of computing like the "Internet of Things"[1]. Frequently, completely new uses for the Web are discovered, often redefining existing beliefs and setting new trends. Amidst all these varying factors, one central factor remains constant—every Web-connected device is bound to communicate with a Web server.

1.1 Motivation

Web servers started out as simple machines delivering static files to a limited number of clients. In just two decades, requirements for Web servers have changed drastically. With the Web becoming ever more volatile and dynamic, fast and reliable server systems are a necessity. For instance, the popular social network *Twitter*[2] handles more than 100000 requests per second on a regular basis—this includes not only static files, but also complex computations and database operations. When designing high-performance server environments, it is favourable to use solutions that utilise the available hardware as efficiently as possible.

However, many proven patterns of software development do not apply to

[1] A network of ubiquitous Web-connected devices, e.g. colour-changing lightbulbs
[2] http://twitter.com

application setups in which a high number of independent operations—like Web requests—has to be handled simultaneously. Such highly *concurrent* operations often require alternative execution concepts and paradigms to be handled more efficiently, or—if the request load exceeds a certain limit—to be processed at all.

1.2 Objective

Currently, two similar alternatives to traditional programming paradigms are used by the industry to increase Web application performance and efficiency: event- and actor-driven paradigms. This thesis aims to give an overview of these technologies by pointing out their characteristic differences and comparing their approaches. Furthermore, current technologies that already implement these paradigms are listed and reviewed in terms of usability and performance in order to analyse the current state of the art. The main goal, however, is to give a clear and educated statement about how and how much event- and actor-based paradigms can increase Web server performance and efficiency and how and to which extent the according technologies can be applied from the view of a developer.

This thesis does not try to define one universal paradigm that fits all general programming needs. It rather tries to find and isolate specific use-cases for certain paradigms with a strong focus on Web server development; focus lies mainly on comprehensible and well-documented technologies that can currently be used by developers without prior experience with the subject.

1.3 Structure

This thesis is divided in two distinct parts: chapters 2 and 3 consist mainly of definitions and research based on existing work, while chapters 4 and 5 almost exclusively contain original research and evaluations.

At the beginning, chapter 2 defines terms, definitions and criteria that are used without further explanation during the remainder of the thesis. Additionally, this chapter provides in-depth explanations of how essential concurrency models are structured and how they utilise system resources. Chapter 3 features a selection of existing event- or actor-based technologies for use with a Web server. For both event- and actor-based paradigms, one solution is reviewed more thoroughly while another solution is portrayed as an alternative and other options are listed briefly. Chapter 4 documents a live project realised by using alternative technologies and elaborates how different technologies were used in order to test and review the applicability of alternative paradigms. Chapter 5 contains a performance evaluation of a traditional application and an actor-driven application in order to define and attest use-cases in which the respective technologies provide better solutions.

Lastly, chapter 6 presents conclusions about the feasibility and applicability of event- and actor-driven paradigms that were elaborated during the course of this thesis. Furthermore, it gives an outlook on the future of event- and actor-driven paradigms and technologies currently in development.

Chapter 2

Technical Background

This chapter aims to give an introduction to common subjects in the area of Web application development and performance analysis. The contents elaborated during this chapter represent a knowledge base that is built upon during the further course of this thesis.

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 a service endpoint communicating with a range of different devices. This section aims to give an overview of the basic requirements a modern Web server 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]; for every request message a client sends to a server, a response message is returned [30]. 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.

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

2.1.2 Dynamic Content

Originally, the Web was intended to be a network of interconnected text files, which later were augmented with images and custom styles; Web servers were basically required to understand incoming requests and respond by sending static content back to the client [30]. With the upcoming of *CGI*[3] during the early 1990s, webpages that are prepared by the server based on dynamic data—like database content or user input were made possible. With the release of *PHP*[4], *ASP*[5] and *Java*[6] in 1995, 1996 and 1997, respectively, dynamic webpages became widespread [31]. From that point on, Web servers needed more processing capabilities for script execution and database access; however, the number of requests per client remained roughly the same [31].

2.1.3 Asynchronous Requests

The advent of *AJAX*[7] and *Web 2.0*[8] in the late 2000's changed requirements drastically. Rather than requiring to refresh the whole view for every piece of information sent and received, data could now be transferred programmatically in the background. By asynchronously communicating with an API[9] endpoint, operations like deleting an item from a list could be performed ubiquitously without reloading the page context. Especially applications that aim to provide desktop-like behaviour and capabilities—commonly called Rich Internet Applications—make heavy use of asynchronous requests [24, p. 4]. Users' expectations for websites changed from anticipating a certain amount of load time to implicating real-time behaviour; to achieve this while maintaining client-server information consistency, latency time must be kept to a minimum. Thus, the server's performance had to meet the combined request frequency of all clients and respond as fast as possible [32].

2.1.4 Request Frequency and Response Time

Since in many cases the responsiveness of the user interface—and with it the user experience—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—

[3] Common Gateway Interface, an interface used to connect the Web server and the actual generated content.

[4] Recursive acronym: PHP Hypertext Preprocessor, http://php.net/

[5] Active Server Pages, http://msdn.microsoft.com/en-us/library/aa286483.aspx

[6] https://www.java.com/

[7] Asynchronous JavaScript and XML (Extensible Markup Language)

[8] A term used to describe the upcoming of dynamic, application-like webpages, often combined with the ability of the users to create and share content.

[9] Application Programming Interface, an interface for connecting different applications at code-level.

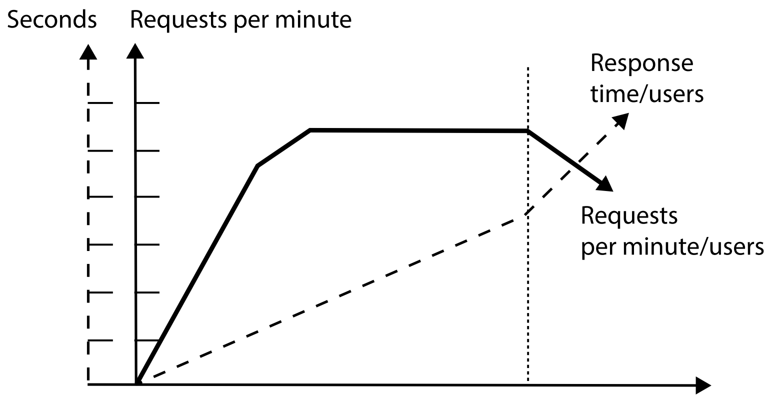


Figure 2.1: Correlation between request frequency and response time in a typical Web server setup. After the server has reached its limit of linearly serving clients (indicated by the dotted line), response times become inversely proportional to the request frequency. Image source: [33].

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 generally become inversely proportional to the request frequency (see figure 2.1). At this point, the server may neglect the request (ideally indicated by returning an error response to the client), not respond at all or even stop serving clients altogether (i.e. "crash") [30].

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 rise or 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 time span [5, 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 across different systems and the load a single unit has to handle decreases; this is described as *scaling out*. If single units are outfitted with more memory and faster processors, i.e. *scaled up*, the number of request operations one unit can process 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 the *pay-what-you-use* model modern service providers offer[10] [17, p. 11]. Software that is well-suited to be expanded according to its usage is considered *elastic* [34].

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[11], scaling out on more than one machine will be unsuccessful [28]. Scalability can be measured by the relationship between hardware resources and the increase of performance. If this relationship is nearly linear, the system is considered to scale well.

2.1.6 Development

Not a part of the production system itself, but nonetheless an essential part of all Web 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 software products can greatly reduce development time and effort, while simultaneously providing robust, tested solutions. Web server applications particularly benefit from frameworks since they often handle standard, repetitive tasks like network I/O[2], database access and caching [23, p. 1]. 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 Web frameworks and their language environments are important criteria.

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 [5, 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, Web applications always have to be implemented using multiple program flows that can be executed concurrently. This section lists various paradigms associated with designing an application capable of maintaining multiple flows of control.

[10] So-called *Cloud* service providers often offer flexible plans on processing power, that can be dynamically adjusted without significant server down-time.

[11] Random Access Memory

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 [35].

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 [28]. 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 or application design [8, 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 [14, p. 162]. This behaviour can be observed for instance in implementations of the popular LAMP[12] server stack configuration [14, p. 48]. A less experienced programmer might find this ideal, since concurrency stays mostly hidden and the application logic is based solely on the flow of a single request—smaller projects might not expose 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

[12] Linux, Apache, MySQL, PHP

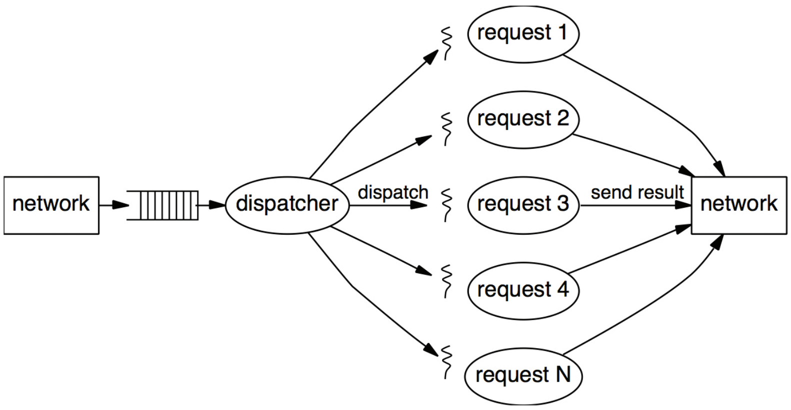


Figure 2.2: On a purely thread-based Web server, each request is handled by a dedicated thread. Incoming network requests are queued and sequentially accepted by a dispatcher, that distributes them among available threads—either by using idle threads from a thread pool or by creating new threads. Image source: [29].

be served—in parallel[13].

Drawbacks

Problems arise when a thread has to wait for an external 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 disk, handling network traffic or file uploads, querying a database, accessing another Web service or processing intensive computations [14, p. 196]. When a thread encounters a blocking operation, it cannot advance further in the program flow until the operation completes (see figure 2.3). The resulting delay can account to anywhere from a few milliseconds to several seconds, for instance when accessing a slow or unresponsive Web service. The only way to counteract the temporary occupation of threads and to continue processing incoming requests is the creation of new threads [17, 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 [36]; 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

[13] 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/).

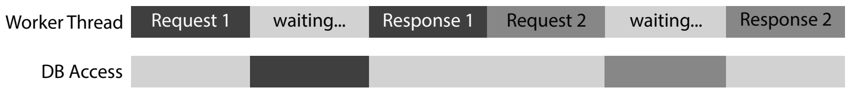


Figure 2.3: A typical blocking situation in Web server scenario. When request one (shown in dark grey) arrives, a database operation is necessary. During the course of this operation, the executing thread blocks while waiting for results (waiting times shown in light grey). The response can only be sent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

threads has to be orchestrated by an operating system module called *scheduler*, which requires processing time relative to the number of threads [36]. 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 [29, 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*[14]—which is common practice due to better replicability—the over-provisioning of memory leads to scarce resources [11, p. 1].

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 [38]. 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 increase linearly in relation to the number of processing cores in a system [39].

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 [26]. Events can be seen as changes in application state; an example of an event may be an arriving HTTP request. An event is often modelled as 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

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

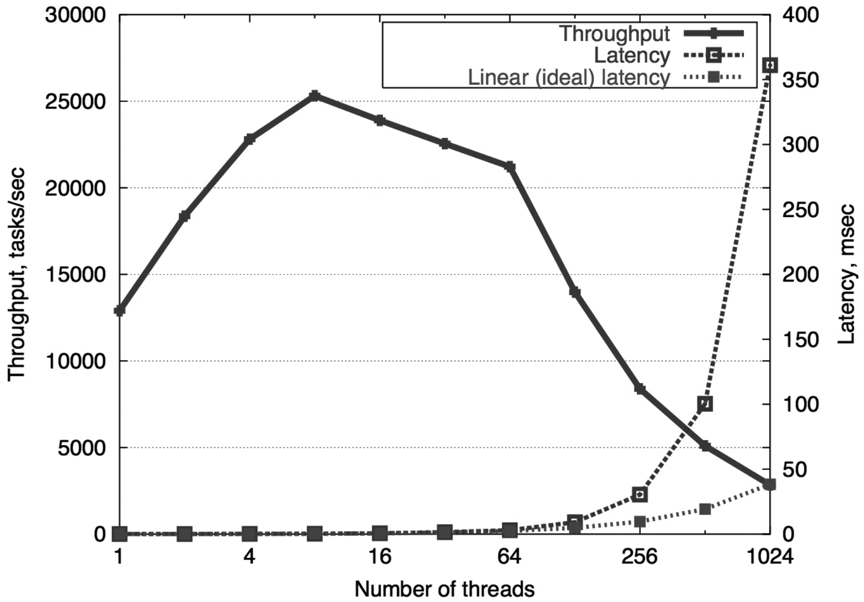


Figure 2.4: This graph shows the performance degradation resulting from rising request frequency for a purely thread-based Web server. Because of the performance overhead introduced by a large number of concurrent threads, the processing throughput decreases. If the number of threads grows above a certain point, the response time escalates due to the shortage of resources. The data is taken from an experiment by M. Welch et al. [29]. Image source: [29].

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 also relinquish control and let itself be controlled by events. This strategy follows the principle of *inversion of control* [16].

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 handled

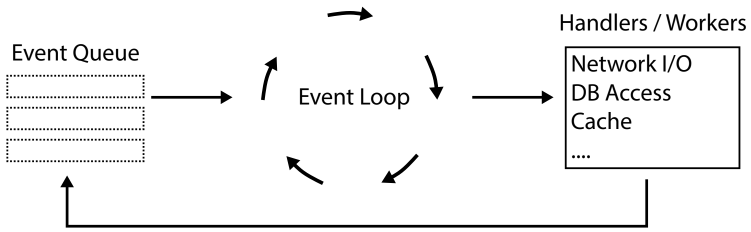


Figure 2.5: Basic flow of control in an event-driven application. Operations that would normally block the event loop are executed separately and create further events upon completion.

by the application.

At its simplest, an event-driven application consists of two major components: on the one hand an *event loop* and on the other hand an *event listener*. 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 and handlers (see figure 2.5) [17, p. 33].

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 another event handler [6, 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 named or anonymous function, as demonstrated in program 2.1.

Callback functions are usually not found in traditional—i.e. sequential—programming. In nearly all application environments, the default way of executing routines in succession is to use functions, which return values that are used during the further course of the program. This proven concept determines three major aspects of program flow [16, p. 3]:

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

Continuation The program flow continues immediately after the function

Program 2.1: Calling a callback function via a named function (above) and via an anonymous 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.

```
function callbackFunction (data) {
    console.log(data);
}

WebService.get("http://example.com/").then(callbackFunction);

```

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

```

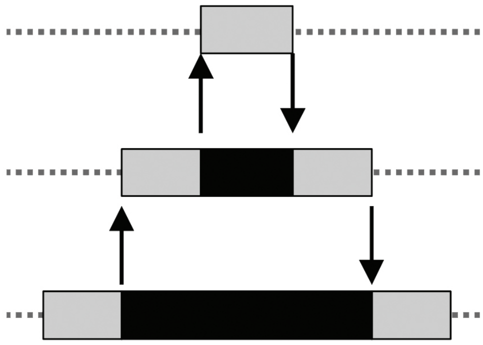


Figure 2.6: Illustration of a simple call stack structure. As time progresses (horizontal axis, left to right), the call stack grows with each function call. As the functions return gradually, the stack size decreases again. Active parts of functions are shown in grey, passive (i.e. "waiting") parts are shown in black. Image source: [16].

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

To store the references and contexts of all functions during such a succession of calls, a dedicated part of memory named the *call stack* is used (see 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 [16].

The departure from the stack 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* [5].

Scalability

In contrast to the purely thread-based concept presented in section 2.2.1, event-based architectures tend to scale better. One major reason for this is the blocking behaviour of the worker 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 (see figure 2.8) [4, p. 2]. Furthermore, because the execution context of event-based program flow is event-specific, global context switching can be minimised. 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[13]. Since context and variables are generally transmitted via immutable events, situations in

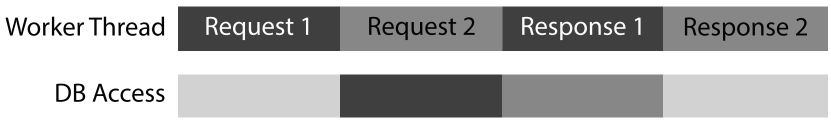


Figure 2.7: A non-blocking scenario in a Web application. When request one (shown in dark grey) arrives, a database operation is necessary. This is a blocking operation, but since the worker thread does not have to wait for it to complete, the next request (shown in medium grey) can already be accepted 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. The figure assumes that all operations take the same amount of time; however, non-blocking operations can of course take an arbitrary amount of time.

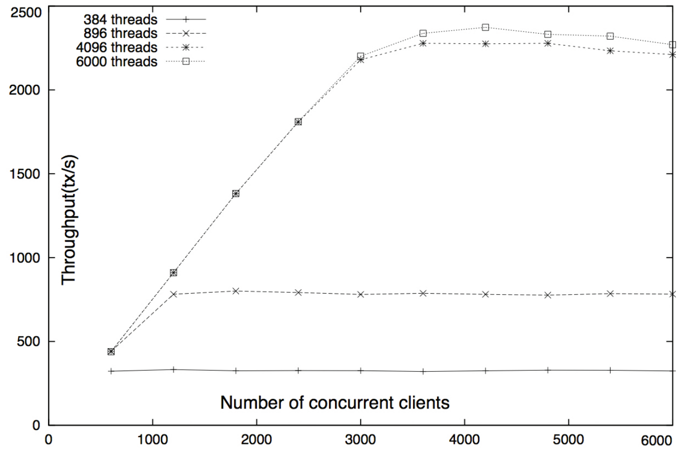
which multiple parts of the program depend and wait on each other (i.e. locking situations) tend to be less common. An example for this is when the same queue is accessed by multiple threads - only one thread can access the queue simultaneously, the other ones have to wait. Furthermore, loose coupling (see above) benefits scalability since parts of the program can easily be replaced and remote procedures can be called more generically.

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 lower 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 benefits the number of parallel requests and thus can lead to significant response time improvements.

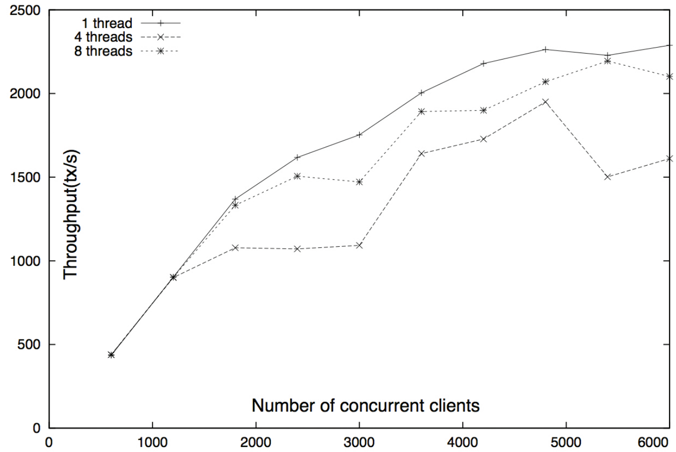
Drawbacks

Beside the implications of relinquishing the call stack pattern—like flat program structure, more or less obfuscated flow of control and reduced state management capabilities—there are other factors that have to be taken into account when using event-driven architectures:

Due to the 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.



(a)



(b)

Figure 2.8: Typical scaling behaviours of a purely thread-based application and an event-based application. This example is taken from an experiment by D. Carrera et al. [4]. The graphs show the effect of an increasing number of clients (x-axis) on the request throughput (y-axis) using a standard *Apache httdp 2.0* Web server—shown in (a)—and using *Java*'s event-based *NIO* interface—shown in (b). Image source: [4].

Additionally, event-based programs lack certain compiler optimisations[15] (given that the programming language used is compiled, rather than interpreted) like advanced memory management, inline functions[16] and compile-

[15]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.

[16]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.

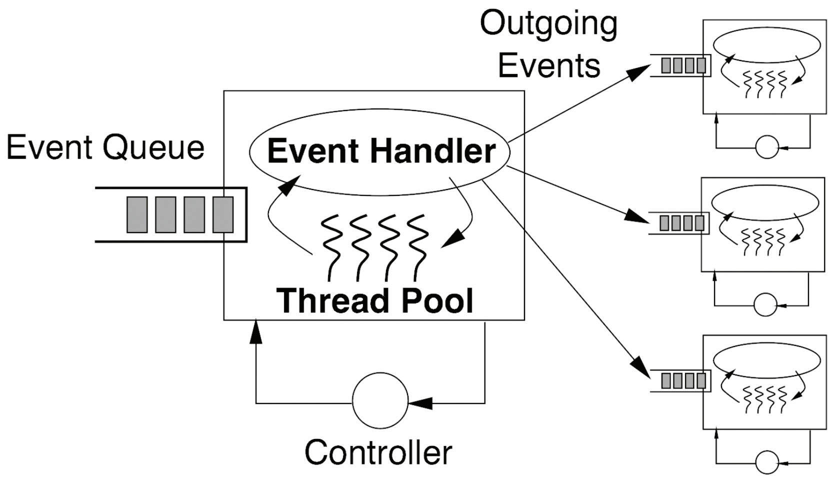


Figure 2.9: Illustration of a single *SEDA* stage (left) and the communication with other stages (right). Each stage has its own event queue, event handler, thread pool and resource controller. Image source: [29].

time warnings about race conditions [2, p. 5].

2.2.3 Staged Event-driven Architecture

Staged event-driven architectures (short *SEDA*) describe a special pattern 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 [29].

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 conditions 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 protection. An application that balances its own resources and changes its behaviour based on parameters like demand and response time is called a *conditioned* application [29].

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* [40]:

> 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 [40]. Apart from that, queues tend to be 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 can 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 [11].

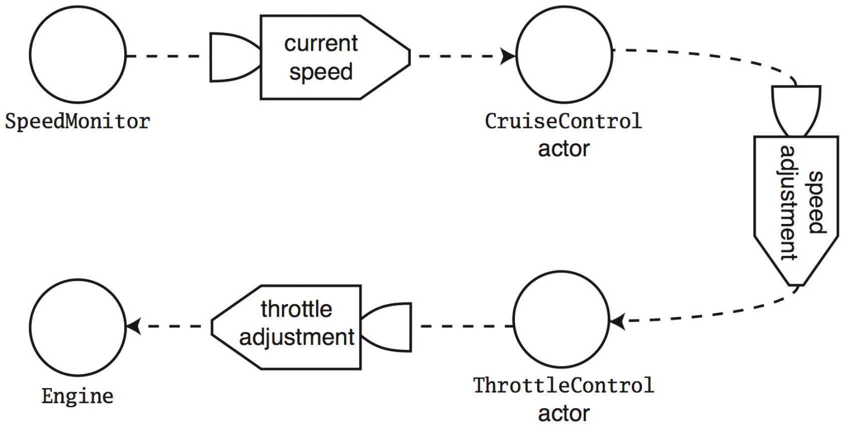


Figure 2.10: Illustration of an actor-based control flow. Actors are represented as circles and messages are represented as rockets. The speed monitor actor periodically sends a message containing the current speed to the cruise control actor. If a deviation of the desired speed is detected, an adjustment message is sent to the throttle control actor, who again sends a message to the engine. Image source: [13].

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 intrinsic 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 [11]. Control remains solely with the sending actor, since it can decide, to whom—if at all—to send further messages (see figure 2.10). This aids in reduced complexity when designing concurrent programs. In this perspective, actor-based architectures can be seen as a 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 in order to create symmetric communication. Furthermore, an actor's internal state can only change in response to incoming messages—there is no way of modifying actor state directly via methods or variables [13, p. 38]. This simple fact has great implications on state management across the application; this architectural characteristic is known as *share-nothing* architecture [3, 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 the mailbox and—since the actor model does not specify a medium for

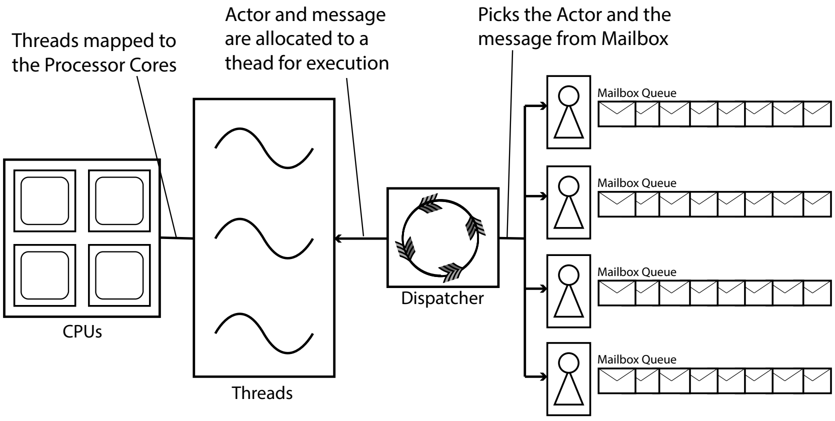


Figure 2.11: This graphic illustrates how resources are handled in a typical actor system. A *dispatcher* distributes messages of actors to operating system threads based on certain strategies (e.g. load balancing). Image source: [10]

.

passing messages (which may also be network-based, see below), a message can take arbitrarily long to reach its destination [6, 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 [7, 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 [10, p. 97]. See figure 2.11 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 supports 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 [12, p. 2]. Moreover, due to their *share-nothing* nature, actors are not limited to one physical system, but work as well on distributed and replicated systems [10, p. 233]. This makes the actor model by far the most scalable of all models 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 generally do not expose the benefits of inheritance and hierarchy to the outside [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 acronym *DRY*[3] [15, p. 149] [22, p. 1].

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. Inbound and outbound network communication represents a large share of common features. 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 persistence is also important for most applications. However, asynchronous I/O and Event- and Actor-based interfaces are the main focus of this chapter. 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. The respective technologies were chosen according to their

[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

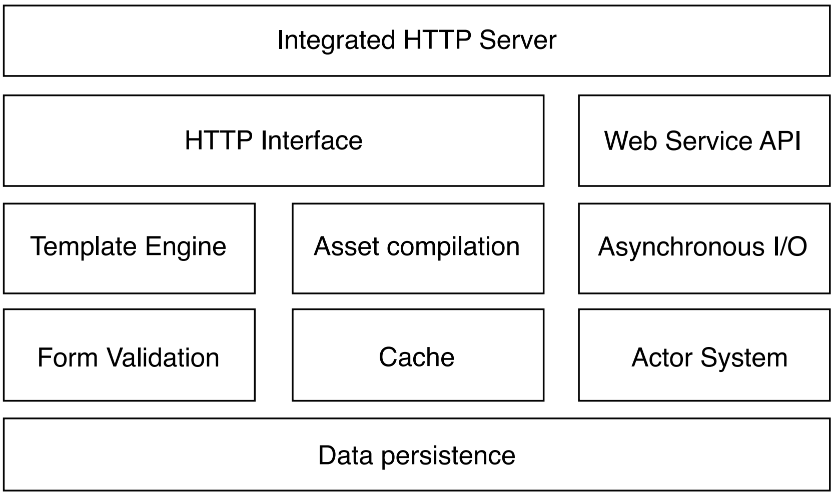


Figure 3.1: A Web framework aims to facilitate development by including frequently used capabilities. Image based on the structure of the *Play! Framework* [15].

current relevancy and popularity in the current technological background and ordered with regard to their applicability in the context of this thesis.

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 the *JavaScript* scripting language and interpreted by the *V8*[5] engine also used in the *Google Chrome* Web browser[6] [17, p. 19]. *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[9].

[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.
[9] http://nodejs.org/api/http.html

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 line 2 calls a function that requests the creation of a new server instance; this function receives an *anonymous* callback function, which is called upon each incoming HTTP request. The function's two parameters are 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: [17, 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 [17, p. 16]. To use a module, it can be included using 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 [42]:

- First-class functions can be handled like any other data type; they can be stored in variables, passed as parameters and executed when needed.
- Parts of the program flow 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 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 the popular *jQuery* library[10] or the *Q* library[11], sequential functions can be written more idiomatically as a chain of commands

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

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: [27, p. 21]

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

```

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: [27, 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
```

(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 (i.e. a specialised syntactic construct) 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[12]. New

[12] http://nodejs.org/api/events.html

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: [43]

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

var ringBell = function ringBell()
{
    console.log('ring ring ring');
}
eventEmitter.on('doorOpen', ringBell);

eventEmitter.emit('doorOpen');

```

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`[13] 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);
```

Independently 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 [6, p. 93]. However, because *JavaScript* is a very popular language due to its use in website development, the fast adoption rate and shallow learning curve of *Node.js* help with building a rich ecosystem around the platform [17, p. 27].

Node.js applications can also benefit from certain framework modules that add *MVC*[14] capabilities. One such module is the *express* framework[15], which includes features such as advanced routing and templates and brings *Node.js* one step closer to being a full-stack Web framework.

[13] https://github.com/caolan/async
[14] Model-View-Controller
[15] http://expressjs.com/

Scalability

Applications running on *Node.js* per default only use a single thread for processing [42]. 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 as well as the 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 [6, p. 105].

To scale out (see section 2.1.5) a *Node.js*-based application, two main steps can be taken: scaling out only 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[16] of *Node.js* (see program 3.5). This way, a master process obtains control over several child processes that handle requests asynchronously based on load balancing [17, 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*[17] systems), the application creates multiple threads in the same process. Running the application on multiple servers has no special implications for *Node.js*; if shared state is desired, it has to be archived by a messaging protocol like *pub-sub* [17, p. 137].

Performance

Node.js is considered very suitable for massive connection concurrency and data-heavy applications [27, p. 44]. The *V8* engine executes *JavaScript* code at a very favourable speed; interfaces that often slow down browser-based applications (like the *DOM*[18]) are not present in a server-side environment. However, since the code is executed via interpretation, its execution is inherently slower than the execution of binary files or virtual machine bytecode. Figure 3.2 illustrates the serious implications of intensive computations on response time.

3.1.2 Eventmachine

Ruby[19] 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 modern Websites [22, p. 11]. However, unlike *JavaScript*, *Ruby* was not con-

[16] http://nodejs.org/api/cluster.html
[17] http://windows.microsoft.com/
[18] Document Object Model
[19] https://www.ruby-lang.org

Program 3.5: The `cluster` module provides an abstraction of creating multiple instances of a program. The first process executing 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: [17]

```
var cluster = require('cluster');
var http = require('http');
var numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
    // Fork workers.
    for (var i = 0; i < numCPUs; i++) {
        cluster.fork();
    }
    ...
} else {
    // Worker processes have a http server.
    http.Server(function(req, res) {
        res.writeHead(200);
        res.end("hello world\n");
    }).listen(8000);
}

```

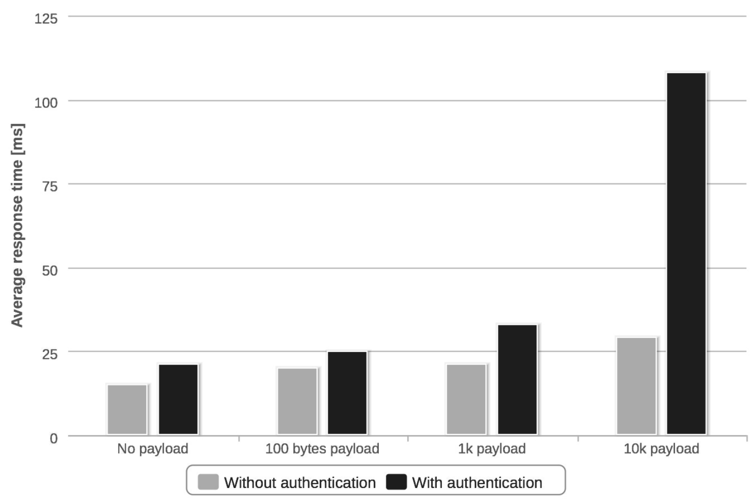


Figure 3.2: In this figure taken from a performance analysis of D. Torstensson and E. Eloff, requests are sent to a *Node.js* server with different payload sizes. The requests are sent both with and without authentication; authentication is done by hashing (i.e. processing) the whole payload using the *SHA1-HMAC* algorithm. Larger payloads are more computationally intensive and result in a longer response time. Image source: [27].

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: [44]

```
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
```

ceived with non-blocking event-driven behaviour in mind. *Eventmachine*[20] 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 [9]. 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[21] 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 [44]). Operations like handling connections and reading input from the sockets also have to be managed explicitly by the developer [44].

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

[20] http://rubyeventmachine.com/

[21] Strictly speaking, since no events are involved, this is called a *reactor loop* [44].

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 achieve *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.

```
operation = proc {
    # long-running operation, e.g. database query
}
callback = proc { |result|
    # do something with result
}

EventMachine.defer(operation, callback)

```

Scalability and Performance

Like *JavaScript*, *Ruby* is an interpreted scripting language and as such performance is inherently inferior to compiled languages[22] (cp. section 3.1.1, *Performance*). When using the default *Ruby* VM[23], a security measure called *Global Interpreter Lock* prevents program threads from archiving physical concurrency by only executing one logical thread at once (see figure 3.3). This is done to prevent sharing non thread-safe code with other threads [45]. 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*[24] is an alternative implementation of the standard *Ruby* interpreter which theoretically allows for controlling physical concurrency at application level [45].

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*[25] is a framework written in *PHP*[26], a scripting language that is often used in simple Web server applications [6, p. 36]. *Twisted*[27] is a reactor library for

[22] 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.
[23] Virtual Machine, a program that executes code inside a dedicated environment.
[24] http://jruby.org/
[25] http://reactphp.org/
[26] https://php.net/
[27] https://twistedmatrix.com/

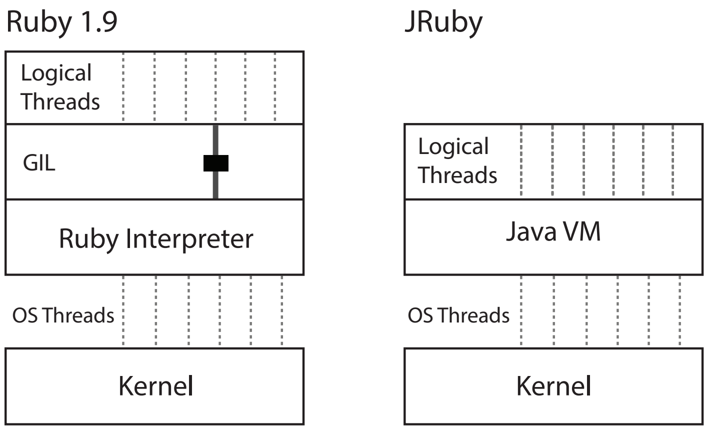


Figure 3.3: The *Global Interpreter Lock* of the *Ruby* VM prevents application logic to control parallel program execution on OS threads because it executes only one *Ruby* thread at once. However, with *JRuby*, this is possible. Image source: [45].

the *Python*[28] scripting language; its capabilities are similar to the *EventMachine* library presented in section 3.1.2. *Java NIO*[29] (see section 2.2.2) is a general interface for non-blocking I/O operations that allows for creating asynchronous Web server applications on a low level.

3.2 Actor-based technologies

3.2.1 Play!

Play![30] is a full-stack Web application framework for development in *Scala*[31] 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* [15]. *Play!* uses the *Akka*[32] actor system, which is—like the *Scala* language and *Play!* itself—managed by *Typesafe Inc.*[33]; all three products are available as an integrated environment called *Activator*.

[28] https://www.python.org/
[29] Native Input and Output
[30] http://www.playframework.com/
[31] http://www.scala-lang.org/
[32] http://akka.io/
[33] 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. Existing since 2003, *Scala* is a rather young programming language that is syntactically very similar to *Java*, but extends it with functional programming capabilities 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 [15, p. 9]; therefore, in this section *Play!*'s functionality is presented using *Scala*.

Play! builds upon the MVC model, which means that incoming requests are handled by a user-defined controller structure. A controller contains *Actions*, i.e. methods that mapped to certain types of requests. 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) [15, p. 86].

Scala provides various language features and libraries for handling concurrency like the `scala.concurrent` library, which includes versatile comprehensions for resolving one or multiple future results. However, *Play!* includes the *Akka* actor system and the respective libraries to even further facilitate

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
```

concurrent processing. *Akka* is used by *Play!* internally for various tasks like request handling, but it can also be used at application-level [15, 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. Program 3.11 demonstrates 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*, 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. 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.

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

Program 3.11: This program is a demonstration of a simple actor used to send emails.

```
case class DefaultMail(
    email: String,
    content: String
)

case class ImageMail(
    email: String,
    image: String
)

class Mailer extends Actor {

    def receive = {
        case DefaultMail(email, content) =>
            sendDefaultMail(email, content)
        case ImageMail(email, image) =>
            sendImageMail(email, image)
    }

    def sendDefaultMail ...

    def sendImageMail ...

}

class Test {

    val mailer = Akka.system.actorOf(Props[Mailer])

    def test() = {
        mailer ! DefaultMail("john@doe.com", "Hello John")
    }

}

```

makes assumptions about the underlying host's thread model [12, p. 3]. The majority of *Play!*'s scalability is due to the included *Akka* actor system [10, 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*:

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 [10, 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 [10, p. 104].

All these 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 [10, p. 233]. This way, *Akka* can be scaled out to a large number of systems.

Performance

Play! internally uses the *Netty*[34] HTTP server, which builds upon *Java NIO* (see section 3.1.3) to achieve non-blocking I/O capabilities [15, p. 52]. While *Netty* has proven to be capable of serving more than 40000 requests per second[35], 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 [46]. 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.

3.2.2 Lattice

Lattice[36] is a lightweight Web framework for the *Ruby* scripting language (see section 3.1.2). It actually represents a combination of several different

[34] http://netty.io/
[35] Tested on an *Amazon EC2* cluster (http://aws.amazon.com/ec2/)
[36] https://github.com/celluloid/lattice

Program 3.12: This program is an adaption of the actor example presented in program 3.11. *Celluloid* actors can be created by simple including the `Celluloid` object inside a *Ruby* class.

```

class Mailer

    include Celluloid

    def send_default_mail(email, content)
        send_mail(...)
    end

    def send_image_mail(email, image)
      send_mail(...)
    end

end

class Test

    mailer = Mailer.new

    def test()

        mailer.async.send_default_mail("john@doe.com, "Hello John")

    end

end

```

technologies that make up the entirety of the framework. *Lattice* uses the *Celluloid* actor system[37] for processing and builds upon the *Reel*[38] Web server. On application level, it uses the *Ruby* port of *Webmachine*[39], which was originally written in *Erlang*[40], to facilitate the handling of HTTP requests by mapping URL routes to respective controller methods [47].

Program 3.12 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. When creating an instance of this class, a new *Celluloid* actor is initialised. To execute an asynchronous routine, the `async` method of the actor object has to be called, followed by the respective method name.

[37] http://celluloid.io/
[38] https://github.com/celluloid/reel
[39] https://github.com/seancribbs/webmachine-ruby
[40] http://www.erlang.org/

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 [48].

3.2.3 Others

Apart from the above frameworks, there are hardly any full-stack Web frameworks that use the actor concurrency model. *Lift*[41] 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*[42] is another *Scala*- and *Akka*-based Web framework that aims to offer functionality similar to *Lift*. *spray*[43] 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.

[41] http://liftweb.net/
[42] http://xitrum-framework.github.io/
[43] http://spray.io

Chapter 4

Implementation

This chapter is a documentation of experiences and remarks 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. Results are used to gain additional 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 in advance:

- The application is a social network. This implies a high request frequency and many atomic database operations.
- There should be client applications for Web as well as for 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.

In order to efficiently process the countless atomic operations that are implied by a social network applications—like interactions between users—as well as the communication to third-party Web services, event- or actor-driven programming paradigms should be heavily used throughout the application.

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

4.1.2 Language and Framework

After defining the prerequisites, the next step is to find a language and a framework that fit the defined needs; since these two elements depend on one another, deciding on a framework inherently limits the choice of languages. 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 and includes a high-performance event-loop, *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 left two choices of language: *Java* and *Scala*. Even though *Scala* is not as widespread as *Java*—which may result in difficulties finding developers in later stages of the project—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[2].

4.1.3 Drivers and Libraries

The present Web application features two kinds of data storage: a *MongoDB*[3] database is used for persisting any long-lived information and a *Redis*[4] 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*[5] due to simple deployment and maintenance. A number of different drivers expose libraries to facilitate communication with these technologies; unfortunately, currently only few drivers 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 *Reac-*

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

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

[4] http://redis.io/

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

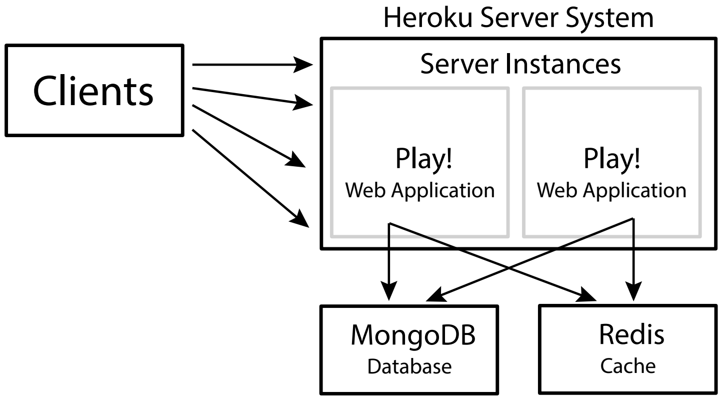


Figure 4.1: The final project setup that resulted from the preceding considerations. Note that client requests are distributed among server instances by the server system, but every server instance connects to the database and cache individually.

tiveMongo[6], a driver implementation written in *Scala* that basically exposes the *MongoDB* API to the application without any additional features like included *DAO*[7] 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*[8] 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. A diagram of the used technologies can be seen in figure 4.1.

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*[9] email implementation written in *Java* can also be used to send emails in *Scala*[10].

[6] http://reactivemongo.org/

[7] Data Access Object, a pattern used by database libraries to simplify storage and retrieval of code objects in the database.

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

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

[10] Blocking *Java* functions can also be wrapped with *Futures* (see sections 3.2.1 and 4.2.2) in order to create non-blocking, actor-based *Scala* functions.

4.2 Development

This section describes how actor-driven patterns were used to realise relevant parts of a Web server application with respect to the prerequisites defined in the last section.

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[11]. 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.

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 4.1 shows an example with two nested `Future` resolutions.

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. This body parser converts the text from the request body into a JSON object[12] 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`

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

[12] JavaScript Object Notation, commonly used for HTTP communication, http://json.org/

Program 4.1: This is a basic example of how two database operations can be nested in a *Play!* application. `Created`, `Conflict` and `InternalServerError` are helpers for the response status codes `201`, `409` and `500`, respectively.

```
def insertUniqueUser() = Action.async(parse.json) {
    request =>
        val username = (request.body \ "username").as[String]
        UserService.findByUsername(username) flatMap {
            case None =>
                UserService.insert(request.body) map {
                    case Some(id) =>
                        Created("New user created with id " + id)
                    case None =>
                        InternalServerError("User could not be created")
                }
             case Some(user) =>
                Future.successful(Conflict("Username exists!"))
        }
}

```

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`, 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.

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 4.2 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: 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 }
```

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:

```
def deleteAlbum(id: String) = Action.async {
    AlbumService.deleteById(id) map {
        case true =>
            ImageService.deleteForAlbum(id)
            Ok("Your album was deleted!")
        case false =>
            InternalServerError("Something went wrong!")
    }
}
```

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:

```
import play.api.libs.concurrent.Akka

Akka.system.scheduleOnce(10.minutes)(sendReminderEmail())

// The first parameter defines the initial delay, the second one the interval
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).

Program 4.3: This configuration statement defines a custom dispatcher called `image-processing-dispatcher` that uses a *fork-join executor* and at most two threads in order not to block the application.

```
akka {
    actor {
        image-processing-dispatcher {
            fork-join-executor {
                parallelism-max = 2
            }
        }
    }
}
```

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:

```
def asynchronousOperation(param: String): Future[String] = {
    Future {
        synchronousOperation(param)
    }
}
```

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 (see program 4.3). Program 4.4 gives an example of an expensive image processing operation converted to an asynchronous operation that can be deferred using a custom dispatcher. At the beginning of the code example, a reference to the custom dispatcher is created using *Akka*'s lookup functionality. The `Future` block 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 consists of a operating system call to a image processing 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.

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

Program 4.4: This program shows how a comparably expensive image processing operation can be deferred using a custom dispatcher.

```
val dispatcher = Akka.system.dispatchers.lookup("akka.actor.image-
    processing-dispatcher")

def generateImage(filename: String): Future[File] = {

    Future {

        val cmd = Array(
            "convert",
            "-background", "black",
            "-fill", "white",
            ...
            filename
        )

        Runtime.getRuntime.exec(cmd).waitFor()

        new File(filename)

    } (dispatcher)

}

```

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 3.11, 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`:

```
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 4.5 shows how this subscriber actor class is structured and used to receive custom *Redis* `publish` messages. Besides the publish channel, *Redis* messages may include a pattern signature; messages are then only distributed to the subscribers 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. 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. A diagram of how information flow happens within the setup can be seen in figure 4.2.

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

[13]Asynchronous Message Queuing Protocol, http://www.amqp.org/

[14]Publish-Subscribe, for *Redis* implementation see http://redis.io/topics/pubsub

Program 4.5: This program shows how the `RedisSubscriberActor` superclass can be used to create an actor class that listens for custom *Redis* `publish` messages.

```
class Subscriber(channels: Seq[String] = Nil, patterns: Seq[String] = Nil) extends RedisSubscriberActor(Redis.socket, channels, patterns, Redis.password) {

    def onMessage(message: Message) {
        Nil
    }

    def onPMessage(message: PMessage) {
        Feeder.push((message.channel, message.data))
    }

}

object Global {

    override def onStart(app: Application) = {
        Akka.system.actorOf(Props(classOf[Subscriber], Nil, Seq("/picture/*/likes")).withDispatcher("rediscala.rediscala-client-worker-dispatcher"))
    }

}

```

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 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 4.5). This means that the application can theoretically be scaled out indefinitely, provided a *load balancer* serves incoming requests fast enough to the server instances

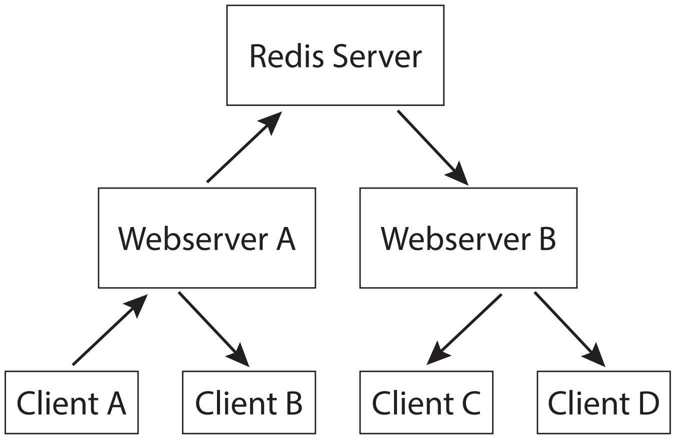


Figure 4.2: Schematic information flow between systems and clients. Client A triggers an event, e.g. likes a picture, which concerns the other three clients. The event is processed by server A, which sends the information directly to all connected clients. Server A *publishes* the event also to the central *Redis* server, which propagates it to all subscribed servers. Server B then sends the received information on to all connected clients.

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).

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 4.1.2 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 due to the high number of *Java*-based Web frameworks. The first choice was the *Grails*[1] framework, which is based on *Groovy*[2] language. Since *Groovy*—like *Scala*—is also an extension language to *Java* and can be run on the *Java* VM, it seemed like a good choice to compare to *Play!*. However, *Grails* proved to have a much larger memory footprint compared to *Play!* and even exceeded the 512MB memory limit on the server container (see below). This imbalance

[1] https://grails.org/
[2] http://groovy.codehaus.org/

ruled out *Grails* as a fitting contender. The *Spring MVC*[3] framework was the next Web framework taken into consideration. *Spring MVC* is regarded as a similar solution compared to *Play!*, especially when it comes to application structure and runtime behaviour [15, p. 109]. As its name implies, *Spring MVC* features a comparable MVC (Model-View-Controller) structure; however, *Spring MVC* does not include its own Web server, but can be executed on a number of different servers. In order to provide similar capabilities as *Play!*'s integrated *Netty* server (see section 3.2.1, *Performance*), *Jetty*[4] was used as a Web server for the *Spring MVC* application.

Obviously, to achieve neutral performance measurements, the two applications must be run on identical, yet independent systems. This, as well as the desire to simulate conditions also present in production setups, led to the decision to conduct the performance tests on virtual servers hosted by a *cloud PaaS*[5]. Due to the simplicity of deployment and replication, *Heroku*[6] was the platform of choice. On *Heroku*, various application technologies—including *Java*—can be deployed and run within isolated containers in a controlled, reproducible fashion. The containers used for testing both are equipped with an *Intel Xeon X5550* quad-core processor clocked at 2.67GHz and can address 512MB of memory.

Both applications were set up using their default configuration and using the *Java* 7 platform. Tests were implemented as controller actions within the MVC environment and were triggered using HTTP requests to certain URLs, exactly as connecting via a website or HTTP interface would. Details on the individual tests are available in section 5.2. To simulate the response of the servers to large client demand, a single personal computer does not suffice. Manual testing, for instance with a browser, can only create several requests per second and even automated testing with tools like *JMeter*[7] may not produce desirable results due to limited bandwidth of a standard internet connections. Thus, a specialised online service that focusses on the simulation of large client loads under realistic conditions was used for testing; *loader.io*[8] offers multiple options of load testing and the free plan supports up to 10000 concurrent connections per test.

5.2 Testing

The first test conducted was a benchmark test to ensure that both systems perform equally in terms of raw computing power. To get an impression of

[3] http://spring.io/
[4] http://www.eclipse.org/jetty/
[5] Platform as a Service. A computing platform provided by a third-party company using large-scale computing and networking systems.
[6] http://www.heroku.com
[7] http://jmeter.apache.org/
[8] https://loader.io/

processing speed, the time to complete an arithmetic integer operation can be measured. Initially, the recursive calculation of a certain number in the *Fibonacci*[9] series was chosen as a suitable calculation. However, the tests produced very different results between the *Scala* and *Java* applications. This is due to the fact that the *Scala* compiler uses a technique called *tail recursion*, which simplifies certain recursive constructs [49]. Thus, an iterative implementation of the *Fibonacci* algorithm was employed with almost equal outcome: the calculation of the 100000th *Fibonacci* number took the *Spring MVC* application 342 milliseconds and the *Play!* application 340 milliseconds on average during ten passes.

After ensuring that both systems work with equal computation speed, the actual load tests could be conducted. *loader.io* offers a test type that gradually increases the request frequency over a set amount of time; in all tests listed here, request frequency was increased during a timeframe of 30 seconds, meaning that each test lasted 30 seconds and then stopped. By measuring the average response time for a certain request frequency, the behaviour of the servers can be characterised. *loader.io* offers a Web-based graph that displays this behaviour for one test per server; however, to acquire the graphs used in this chapter, a dedicated script gathered the different datasets and merged them to graphs that display the data for both applications with equal scale.

There are two main test categories. The first observes the behaviour of the servers for "on-system operations", i.e. calculations that demand computation time directly on the server—like for instance image processing. The second category observes "off-system operations" like Web requests, database queries or cache lookups. Both categories feature a number of tests with varying parameters to generate different benchmarks of the server behaviour. During all tests, parameters were chosen in a way that generated interesting and meaningful results, but without compromising server stability; load was limited to the maximum both servers could handle without crashing.

On-system tests simulated server load by calculating a certain *Fibonacci* number for each request, thus blocking either the Web thread (in the *Spring MVC* application) or an actor-based worker thread (in the *Play!* application). The complexity of the calculation was increased with different step sizes (with the linear increase rate of the request frequency being constant) and the most interesting results were chosen for presentation in the following section.

Off-system tests loaded the content of a website[10], which was chosen because of its resilience) and tunnelled it through the server applications to the client. This *proxy*[11]-like behaviour simulates operations that happen outside

[9] *Fibonacci* numbers are produced by adding up the previous two numbers of the series, starting with 0 and 1.

[10] http://www.google.com was used due to its resilience.

[11] A proxy server is a system that transmits network requests back and forth between

of the server system and block only threads on purely thread-based applications, while not consuming threads in event- and actor-driven applications. Since an increase in website size did not make much difference for the tests, the website to load was constant and the request frequency increase was increased with each test. Again, the most notable results were chosen for presentation.

5.3 Results

On-system Tests

The first test was set up using the relatively fast calculation of the 1000th *Fibonacci* number. A single request of this kind took both applications on average 170 milliseconds to process. However, with increasing request frequency, the average response times of the two applications drift apart with the *Play!* application achieving generally lower response times, the difference being about 1000 milliseconds at 5000 requests per second (see figure 5.1). However, when increasing the computation complexity by calculating the 5000th *Fibonacci* number—resulting in an average response time of 300 milliseconds per request—the response time difference between the two applications decreases with *Play!* still being faster, but only with a difference of about 500 milliseconds at 5000 requests per second (see figure 5.2). A further increase of the computation complexity to the 10000th and 50000th *Fibonacci* number—resulting in 300 and 400 milliseconds average response time per request, respectively—shows that the response times of the two applications become more and more similar until being nearly identical for the 50000th *Fibonacci* number (see figures 5.3 and 5.4).

Off-system Tests

As already mentioned in section 5.2, off-system tests were conducted by loading the content of the main page of *Google*[12], which, at the time of testing, contained 44 kilobytes of data and took about 170 milliseconds to load. Starting at a request frequency of 0 to 100 requests per second, the *Play!* application outperforms the *Spring MVC* application from about 30 requests per second and maintains low response times until the end of the test, while the response times of the *Spring MVC* application continue to rise more steeply (see figure 5.5). Increasing the request frequency to 1000 exhibits even more pronounced results with the *Play!* application being slower at the beginning, but from about 180 requests per second on maintaining a response time of about 800 milliseconds while the *Spring MVC* application becoming ever slower until the difference accounts to about 2400 milliseconds at 1000

client and target. This has different uses, for instance increased anonymity.

[12] http://www.google.com

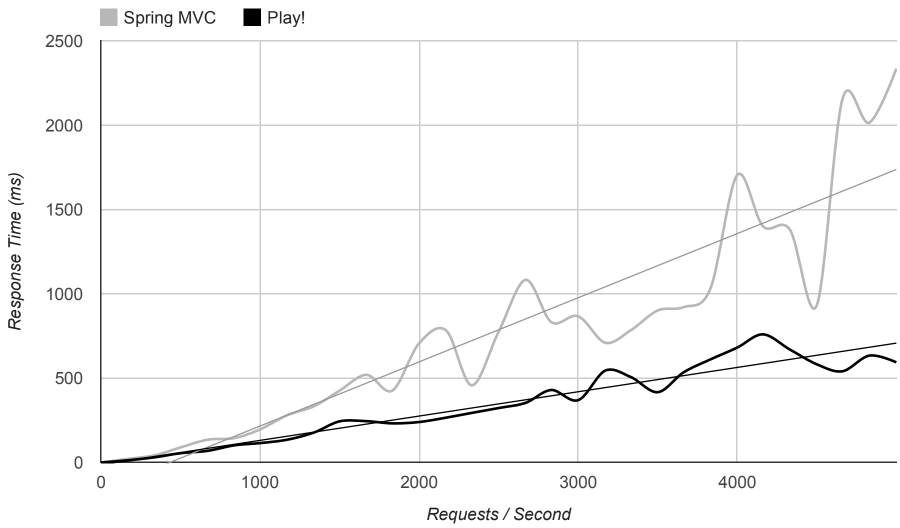


Figure 5.1: Iterative calculation of the 1000th *Fibonacci* number. Request frequency is increased to 5000 within 30 seconds.

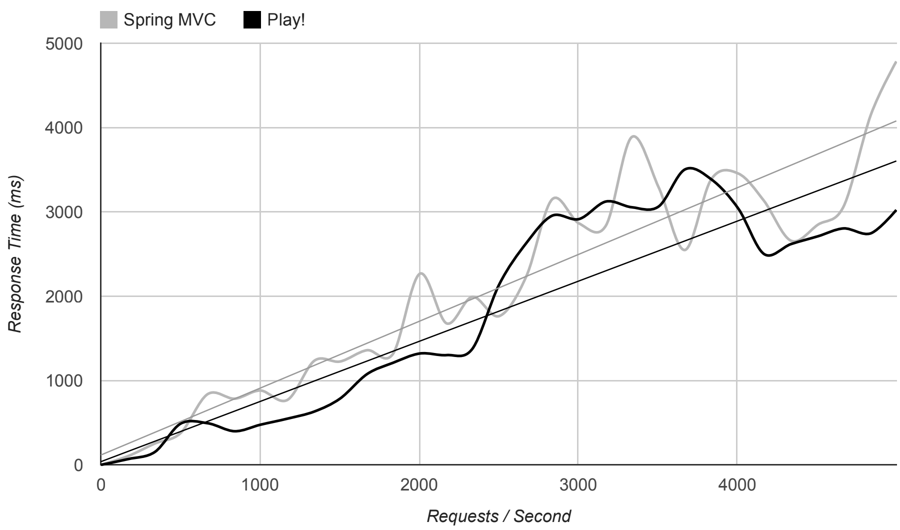


Figure 5.2: Iterative calculation of the 5000th *Fibonacci* number. Request frequency is increased to 5000 within 30 seconds.

requests per second (see figure 5.6). During the third test of this category, request frequency is increased to 5000, obviously putting a lot of pressure on the applications. The response graph for this test looks very similar to the first graph of the on-system category (see figure 5.1) with response times

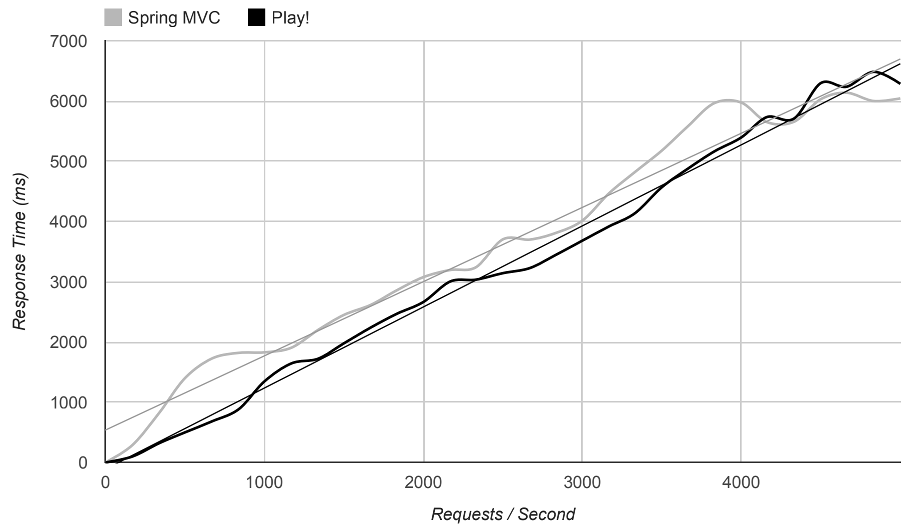


Figure 5.3: Iterative calculation of the 10000th *Fibonacci* number. Request frequency is increased to 5000 within 30 seconds.

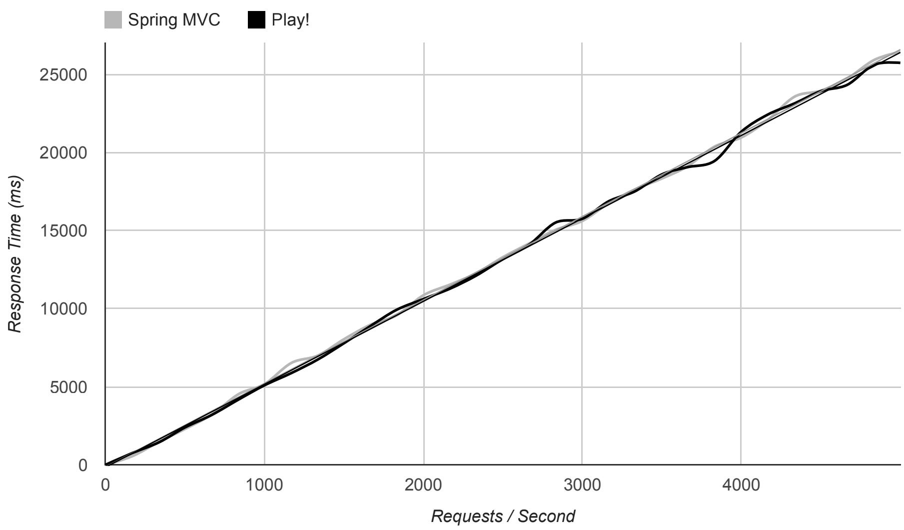


Figure 5.4: Iterative calculation of the 50000th *Fibonacci* number. Request frequency is increased to 5000 within 30 seconds.

of both applications increasing continuously at a high rate and the *Play!* application being about 3800 milliseconds faster (see figure 5.7).

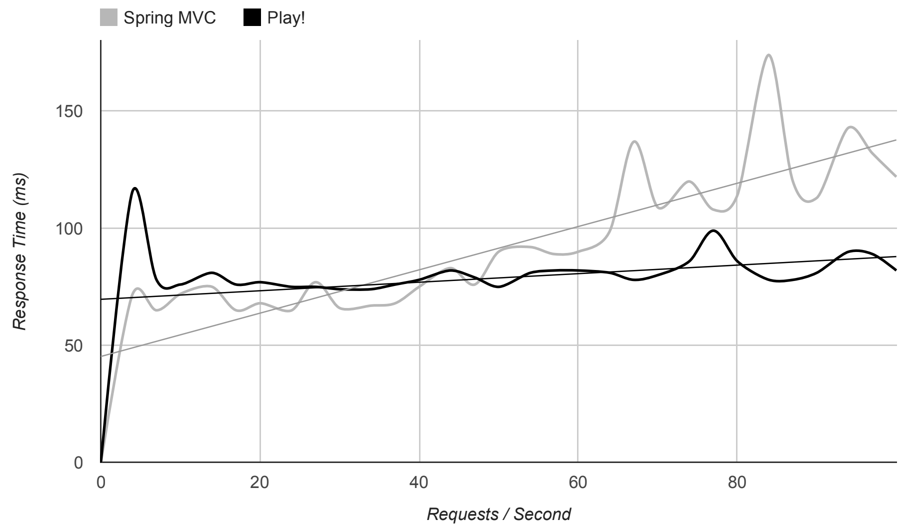


Figure 5.5: Loading and serving an off-system website. Request frequency is increased to 100 within 30 seconds.

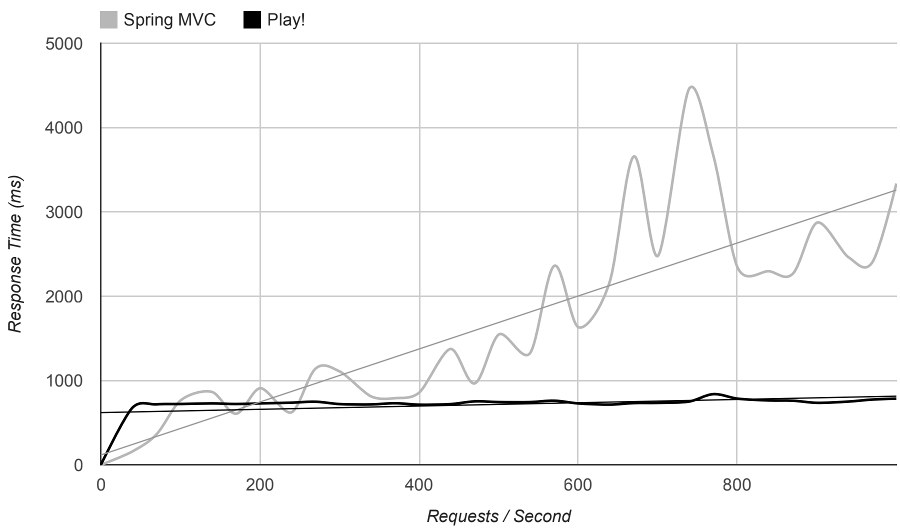


Figure 5.6: Loading and serving an off-system website. Request frequency is increased to 1000 within 30 seconds.

5.4 Interpretation

Looking at the tests, it is obvious that there is a significant difference between the on-system and off-system tests. In the on-system scenario, the per-

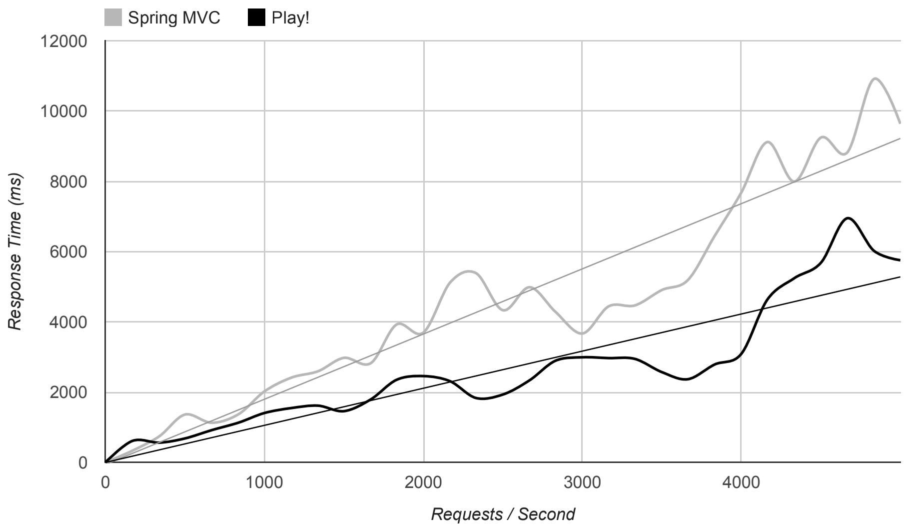


Figure 5.7: Loading and serving an off-system website. Request frequency is increased to 5000 within 30 seconds.

formance difference between the blocking and the non-blocking application decreases with increasing server load while in the off-system scenario, the performance difference increases. Regarding the concurrency models detailed in section 2.2, this behaviour can be explained with the different resource usage on the servers. The context-switching overhead of the purely thread-based *Spring MVC* application leads to a slower performance at a lower request frequency, while the actor-based *Play!* application employs worker threads and reuses the same threads for network communication and request handling. However, with increasing computation complexity, the processing resources of both systems tend to be consumed wholly, which practically annihilates any advantage of lower context-switching overhead and worker threads; since both systems use identical hardware, both processors are saturated at the same point.

With off-system operations, on the other hand, the outcome is completely different. While the *Play!* application achieves slower response times at the beginning due to the computational intensity of maintaining and orchestrating the actor system, it does not use up threads while waiting on the answer of the off-system Web server. Thus, up to a certain point, the *Play!* application can maintain almost equal response times even with an increasing request frequency. The *Spring MVC* application does not disengage the current thread during waiting for the response; this effectively blocks the current thread, raising the need to create or use another thread for the next incoming request. This puts more pressure on the operating system scheduler and creates contest-switching overhead. Moreover, since the processors used have four cores, only four threads can achieve physical concurrency at once; therefore, the *Play!* application can use resources more efficiently by at all times doing active computations on all cores. However, when the request frequency exceeds the speed with which the *Play!* application can handle communication with the off-system Web server, response times increase with actor queue size. This again creates a scenario in which the response time is directly proportional to the request frequency, much like with non-saturated on-system operations.

Chapter 6

Conclusion and Future Development

Developing event- and actor-driven applications still poses some difficulties. The purely thread-based concurrency model seems like a natural abstraction choice for developing Web server applications, but problems like blocking and context-switching overhead make event- and actor-based operations the better choice for most Web-typical use-cases. On the other hand, although event- and actor-based paradigms have existed for decades, they represent a significant departure from traditional thread-based programming and thus may seem a less attractive choice for developers. To approach this problem, modern Web frameworks and libraries offer simple abstractions for complex operations; these range from concurrency comprehensions to system-wide abstractions like completely hiding the underlying actor system, like the *Play!* framework does. Many of these frameworks also offer thread-based processing, which further minimises the learning curve. *Node.js* makes use of *JavaScript* and its event-driven development paradigms, which are known to even less experienced Web developers.

Regardless of the technology used, when developing applications with a high amount of concurrency, close attention has to be paid to maintain a clear application structure. Modern Web frameworks and libraries often facilitate this process by offering simplifications and structural rules and guides. However, purely thread-based have the advantage of ordered execution, which results in a structural clarity that is hard to obtain with event- and actor driven development.

It can be concluded from the tests documented in chapter 5 that asynchronous processing reduces response times for most Web server specific tasks like querying databases and caches as well as communicating with third-party Web services. However, most of these advantages come into effect only above a certain request frequency. Projects that tend to benefit the most from asynchronous processing are characterised by a high number of computationally inexpensive requests involving database or networking operations; a prominent example for this use-case are social networks, where a lot of users use the same platform simultaneously. For smaller projects like personal websites, on the other hand, purely thread-based applications may be the better choice since they can be developed with traditional paradigms in a widespread language like *PHP*. Another use-case for thread-based applications are systems that focus on intensive on-system calculations at a high request frequency. Here, the best strategy may be to use a highly synchronous architecture and scaling out to multiple systems.

Especially with the uprise of more frameworks and libraries, event- and actor-driven paradigms currently are gaining popularity and importance. There is an initiative called the *Reactive Manifesto*, which advocates asynchronous, non-blocking behaviour throughout Web applications, even including client applications [34]. Many emerging as well as older technologies are adding asynchronous functionality in order to offer modern functionality. Such technologies include the asynchronous additions made to *C#*[1] in version 5, the *Grails* 2.3 events API and the `@Async` annotated methods in *Spring MVC* 3 [25] [18]. Another notable example is the *Martini*[2] framework for the *Go*[3] language, which uses extensive asynchronous processing featuring a concurrency model called *Pipelines*.

Even though there are numerous new technologies, only few mature Web frameworks already make use of them, despite potentially large performance gains for exactly these systems. For instance, *Node.js* has a shallow learning curve and exceptional performance and documentation, but lacks application structure and compile-time warnings. Many other frameworks that were taken into consideration for this thesis lacked either proper functionality or essential documentation. However, the high number of event- and actor-driven Web frameworks currently in early stages of development gives a positive outlook on the future progress of the subject. With demands and user numbers ever growing, the Web is merely at the beginning of reaching its full potential. Event- and actor-driven paradigms—if sufficiently embraced by Web developers—can play a major role in creating a fast, efficient Web for the future.

[1] http://msdn.microsoft.com/en-us/vstudio/hh341490.aspx
[2] http://martini.codegangsta.io/
[3] http://golang.org/

Appendix A

Contents of the CD-ROM

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 Master Thesis

Path: /

Hessenberger_Felix.pdf A digital version of this document in PDF (Portable Document Format) format

A.2 Online Sources

Path: /Online Sources

*.pdf . . . . Digital versions of the used online sources, named according to citation numbers

A.3 Test Applications

Path: /Test Applications

play/* . . . . Source code of the *Play!* application from chapter 5

spring/* . . . . Source code of the *Spring MVC* application from chapter 5

References

Literature

[1] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[2] Rob Von Behren, Jeremy Condit, and Eric Brewer. "Why Events Are A Bad Idea (For High-Concurrency Servers)". In: *HotOS*. Lihue, Hawaii, USA, 2003, pp. 19–24.

[3] Daniele Bonetta, Danilo Ansaloni, and Achille Peternier. "Node.Scala: Implicit Parallel Programming for High-Performance Web Services". In: *Euro-Par*. Rhodes Island, Greece, 2012, pp. 626–637.

[4] David Carrera et al. "Evaluating the Scalability of Java Event-Driven Web Servers". In: *International Conference on Parallel Processing*. Montreal, Canada, 2004, pp. 134–142.

[5] Vaarnan Drolia, Cedric Ansley, and Chin Shen. "Threads vs Events for Server Architectures". Survery Paper, National University of Singapore, 2010.

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

[7] Joakim Eriksson. "Representation of Asynchronous Communication Protocols in Scala and Akka". MA thesis. Linköping University, 2013.

[8] Jeffrey Fischer, R Majumdar, and Todd Millstein. "Tasks: Language Support for Event-Driven Programming". In: *ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation*. Nice, France, 2007, pp. 134–143.

[9] Michael Fitzgerald. *Learning Ruby*. O'Reilly Media, Inc, 2007.

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

[11] Philipp Haller and Martin Odersky. "Event-Based Programming Without Inversion of Control". In: *Joint Modular Languages Conference*. Oxford, UK: Springer-Verlag Berlin, 2006, pp. 4–22.

[12] Philipp Haller and Martin Odersky. "Scala Actors: Unifying Thread-based and Event-based Programming". In: *Theoretical Computer Science* 410.2 (Feb. 2009), pp. 202–220.

[13] Philipp Haller and Frank Sommers. *Actors in Scala*. artima Press, 2011.

[14] Cal Henderson. *Building Scalable Web Sites*. O'Reilly Media, Inc, 2006.

[15] Peter Hilton, Erik Bakker, and Francisco Canedo. *Play for Scala: Covers Play 2*. Manning Publications Co., 2013.

[16] Gregor Hohpe. "Programming Without a Call Stack - Event-Driven Architectures". In: *Object-Oriented Programming, Systems, Languages and Applications*. San Diego, California, USA, 2005, pp. 12–21.

[17] Tom Hughes-Croucher and Mike Wilson. *Node - Up and Running*. O'Reilly Media, Inc, 2012.

[18] Peter Ledbrook. "Application Architectures in Grails". In: *GR8Conf US*. Minneapolis, Minnesota, USA, 2014, pp. 32–48.

[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*. London, UK, 1997, pp. 20–23.

[21] Sucheta Nadimpalli and Shikharesh Majumdar. "Techniques for Achieving High Performance Web Servers". In: *International Conference on Parallel Processing*. Toronto, Canada, 2000, pp. 233–242.

[22] Rob 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". Working Paper, 2011.

[25] Don Syme, Tomas Petricek, and Dmitry Lomov. "The F# Asynchronous Programming Model". In: *Practical Aspects of Declarative Languages*. Austin, Texas, USA: Springer-Verlag Berlin, 2011, pp. 175–189.

[26] Stefan Tilkov and Steve Vinoski Verivue. "Node.js : Using JavaScript to Build High-Performance Network Programs". In: *Internet Computing, IEEE* 14.6 (2010), pp. 80–83.

[27] Daniel Torstensson and Erik Eloff. "An Investigation Into the Applicability of Node.js as a Platform for Web Services". MA thesis. Linköping University, 2012.

[28] Bryan Veal and Annie Foong. "Performance Scalability of a Multi-Core Web Server". In: *Architecture for Networking and Communications Systems*. Princeton, New Jersey, USA: ACM Press, 2007, pp. 57–66.

[29] Matt Welsh, David Culler, and Eric Brewer. “SEDA : An Architecture for Well-Conditioned, Scalable Internet Services”. In: *Symposium On Operating Systems Principles*. Banff, Canada: ACM Press, 2001, pp. 230–243.

Online Sources

[30] URL: http://www.ietf.org/rfc/rfc2616.txt (visited on 03/30/2014).

[31] URL: http://royal.pingdom.com/2007/12/07/a-history-of-the-dynamic-web/ (visited on 02/21/2014).

[32] URL: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/ (visited on 02/21/2014).

[33] URL: http://docs.oracle.com/cd/E19900-01/819-4741/fygaj/index.html (visited on 03/30/2014).

[34] URL: http://www.reactivemanifesto.org/ (visited on 09/21/2014).

[35] URL: http://www.nakov.com/inetjava/lectures/part-1-sockets/InetJava-1.3-Multithreading.html (visited on 03/02/2014).

[36] URL: http://blogs.technet.com/b/markrussinovich/archive/2009/07/08/3261309.aspx (visited on 03/02/2014).

[37] URL: http://mmcgrana.github.io/2010/07/threaded-vs-evented-servers.html (visited on 03/30/2014).

[38] URL: http://www.ibm.com/developerworks/java/library/j-jtp0730/index.html (visited on 03/30/2014).

[39] URL: http://software.intel.com/en-us/blogs/2008/06/05/nitrogen-narcosis-part-ii-the-serious-drawbacks-of-explicit-multi-threading/ (visited on 03/02/2014).

[40] URL: http://matt-welsh.blogspot.co.at/2010/07/retrospective-on-seda.html (visited on 03/30/2014).

[41] URL: http://thornydev.blogspot.co.at/2012/01/events-and-event-driven-architecture.html (visited on 03/11/2014).

[42] URL: http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/ (visited on 07/03/2014).

[43] URL: http://www.sitepoint.com/nodejs-events-and-eventemitter/ (visited on 06/12/2014).

[44] URL: https://dl.dropboxusercontent.com/u/635/em_export.pdf (visited on 07/13/2014).

[45] URL: https://www.igvita.com/2008/11/13/concurrency-is-a-myth-in-ruby/ (visited on 07/28/2014).

[46] URL: http://blog.papauschek.com/2013/04/real-world-performance-of-the-play-framework-on-ec2/ (visited on 08/08/2014).

[47] URL: https://github.com/celluloid/celluloid/wiki/Basic-usage (visited on 08/09/2014).

[48] URL: https://github.com/celluloid/celluloid/wiki/Futures (visited on 08/08/2014).

[49] URL: http://oldfashionedsoftware.com/2008/09/27/tail-recursion-basics-in-scala/ (visited on 08/21/2014).