

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

FELIX HESSENBERGER

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

INTERACTIVE MEDIA

in Hagenberg

im Juni 2014

© Copyright 2014 Felix Hessenberger

This work is published under the conditions of the *Creative Commons License Attribution–NonCommercial–NoDerivatives* (CC BY-NC-ND)—see <http://creativecommons.org/licenses/by-nc-nd/3.0/>.

Declaration

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

Hagenberg, June 30, 2014

Felix Hessenberger

Contents

Declaration	iii
Abstract	vi
Kurzfassung	vii
1 Introduction	1
1.1 Motivation	1
1.2 Objective	3
1.3 Structure	4
2 Technical Background	5
2.1 Terms and Definitions	5
2.1.1 Network Communication	5
2.1.2 Dynamic Content	5
2.1.3 Asynchronous Requests	6
2.1.4 Request Frequency and Response Time	6
2.1.5 Scalability	7
2.1.6 Development	8
2.2 Concurrency Models	8
2.2.1 Purely Thread-based Model	8
2.2.2 Event-based Model	12
2.2.3 Staged Event-driven Architecture	18
2.2.4 Actor Model	19
3 State of the Art	22
3.1 Event-based technologies	23
3.1.1 Node.js	23
3.1.2 Eventmachine	27
3.1.3 Others	30
3.2 Actor-based technologies	31
3.2.1 Play!	31
3.2.2 Lattice	36
3.2.3 Others	38

4 Implementation	39
4.1 Project Setup	39
4.2 Development	40
4.3 Deployment	43
4.4 Maintenance	45
5 Evaluation	48
5.1 Approach	48
5.2 Performance	51
5.3 Maintenance	55
5.4 Results	57
6 Conclusion and Future Development	60
References	63
Literature	63
Online Sources	64

Abstract

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

Kurzfassung

Franz rast im komplett verwahrlosten Taxi quer durch Bayern.

Chapter 1

Introduction

1.1 Motivation

1. Introduction

2

a

a

1.2 Objective

a

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

1.3 Structure

Chapter 2

Technical Background

2.1 Terms and Definitions

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

2.1.1 Network Communication

The eponymous task of a Web server is to serve Web-connected clients over the medium of the Internet. This involves receiving and sending messages using different implementations of network protocols. The most widely used protocol of the Web, *HTTP*¹, is a request-response protocol, which means that for every message a client sends to a server, a response is sent back [30]. To minimise networking latency, it is preferable for a Web server to have a high-speed connection to the Internet, fast system I/O² and capable routing hardware. However, these parameters are not directly related to software and are thus neglected during the further course of this thesis.

2.1.2 Dynamic Content

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

¹Hypertext Transfer Protocol

²Input and Output, esp. hardware

content accordingly [30]. With the release of *PHP*³, *ASP*⁴ and *Java*⁵ – in 1995, 1996 and 1997, respectively – dynamic webpages, i.e. views that are prepared by the server based on dynamic data like database content, became widespread [39]. From that point on, Web servers needed more processing capabilities for script execution and database access; however, the number of requests remained roughly the same, except for occasional form submissions [39].

2.1.3 Asynchronous Requests

The advent of *AJAX*⁶ and mobile applications in the late 2000's changed requirements drastically. Rather than refreshing the whole view for every piece of information sent and received, data could now be transferred in a more granular fashion. By asynchronously communicating with an API⁷ endpoint in the background, operations like deleting an item from a list could be performed invisibly and ubiquitously without reloading the page context. Especially applications that aim to provide desktop-like capabilities – commonly called Rich Internet Applications – make heavy use of asynchronous requests [24, p. 4]. This inherently also changed users' expectations for websites from anticipating a certain amount of load time to implicating real-time behaviour [31]. To achieve low latency while maintaining client-server information consistency, the server's performance has to meet the combined request frequency of all clients at a given point in time.

2.1.4 Request Frequency and Response Time

Since in many cases the responsiveness of the user interface depends on the duration of the server communication roundtrip, maintaining acceptable response times is often crucial [21, p. 1]. Request frequency and response time correlate in the sense that request frequency represents the demand on a server endpoint while response time – given equally demanding operations per request – can be interpreted as the potential of the server to meet the demand. When the processing limit of the server is met, response times become generally inversely proportional to the request frequency, as illustrated in figure 2.1 [37]. At this point, the server may chose to neglect the request (ideally by returning the status code *503 Service Unavailable* [30]), not respond at all or even stop serving clients altogether (i.e. “crash”).

³Recursive acronym: PHP Hypertext Preprocessor, <http://php.net/>

⁴Active Server Pages, <http://msdn.microsoft.com/en-us/library/aa286483.aspx>

⁵<https://www.java.com/>

⁶Asynchronous JavaScript and XML (Extensible Markup Language)

⁷Application Programming Interfaces

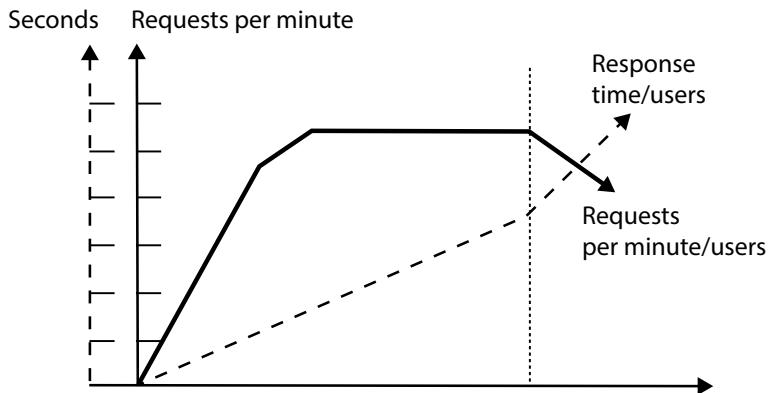


Figure 2.1: Correlation between request frequency and response time in a typical Web server setup. After the server has reached its limit of linearly serving clients (indicated by the dotted line), response times become inversely proportional to the request frequency. Image source: [37]

2.1.5 Scalability

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

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

Ideally, the server software should be hardware agnostic, i.e. should behave consistently independent of the hardware it runs on. For instance, if the software depends heavily on sharing application state via RAM⁹, scaling out on more than one machine will be unsuccessful [27]. Scalability can be measured by the relationship between hardware resources and the increase

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

⁹Random Access Memory

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

2.1.6 Development

Not a part of the production system itself, but nonetheless an essential part of all Web server applications is their development. A structured, idiomatic way of writing application logic doubtlessly contributes to every software product. Modularisation of components facilitate the use of third-party software like libraries and frameworks. In return, using existing solutions can greatly reduce development time and effort, while simultaneously providing proven solutions. Web server applications particularly benefit from frameworks since they often handle standard tasks like network I/O², database access and caching [23, Foreword]. Integrating and maintaining these frameworks is a major part in implementing a Web server application; thus, not only the performance, but also the ease of use of selected frameworks and their language environments by the developer are treated as criteria in this thesis.

2.2 Concurrency Models

Since a Web application in a production setting is usually publicly accessible, serving multiple clients simultaneously is the rule, rather than the exception. Depending on the popularity of the service, the number of concurrent requests can range anywhere from dozens to several thousands, e.g. for social media sites [6, p. 1]. A server process with a single flow of control would only be able to serve one client at once, with all requests received while the server is busy being neglected. Therefore, networking applications always have to be implemented using multiple program flows that can be executed concurrently [42]. This section lists various paradigms associated with designing an application capable of maintaining multiple flows of control.

2.2.1 Purely Thread-based Model

A thread is a sequence of instructions within a program. Allocating processing time to threads is handled by an operating system scheduler. To have a program execute multiple logic structures concurrently, they have to be explicitly abstracted in the form of threads. Physical concurrency occurs, when threads are executed simultaneously – i.e. at the exactly same time – on different processor cores; in contrast, logical concurrency describes that multiple threads are executed sequentially in rapid succession at roughly the same time, thus giving the impression of simultaneous execution. Physical concurrency is inherently more efficient [36].

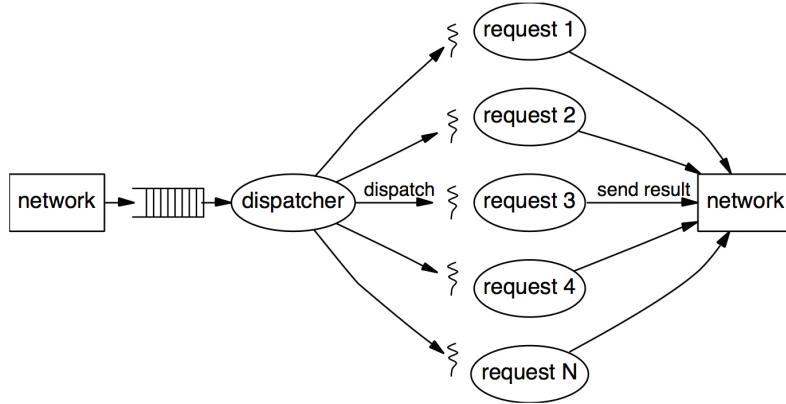


Figure 2.2: On a purely thread-based Web server, each request is handled by a dedicated thread. Incoming network requests are queued and sequentially accepted by a dispatcher, that distributes them among available threads – either by using idle threads from a thread pool or by creating new threads. Image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

Flow of Control

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

Scalability

Traditionally, Web server applications process each request on a dedicated thread throughout its whole lifespan, from accepting it to responding to it [15, p. 162]. This behaviour can be observed for instance in implementations of the popular LAMP¹⁰ server stack configuration [15, p. 48]. A less experienced programmer might find this ideal, since concurrency stays mostly

¹⁰Linux, Apache, MySQL, PHP

hidden and the application logic is orientated on the flow of a single request – smaller projects might not experience any drawbacks of this setup at all. However, it is obvious that to scale up a thread-based system the number of threads has to be increased. The number of threads engaging in simultaneous processing, i.e. physical concurrency, is limited by the number of processing cores. This means that on a computer equipped with a quad-core processor, four threads can be executed – and thus, four requests can be served – in parallel¹¹.

Drawbacks

Problems arise when a thread has to wait for another requirement to be fulfilled. The process of meeting a requirement that renders the executing thread unable to proceed is called a *blocking* operation. Such actions include for instance reading or writing a file on mass storage, handling network traffic or file uploads, querying a database, accessing another Web service or doing intensive computations [15, p. 196]. When a thread encounters a blocking operation, it cannot advance further in the program flow until the operation completes, as illustrated in figure 2.3. The resulting delay can account to anywhere from a few milliseconds to several seconds, for instance when accessing a slow or 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 [18, p. 36]. However, every newly created thread counts towards certain limitations in scalability. On the one hand every thread receives a predefined share of process address space memory – also known as *stack* – upon creation to temporarily store data [40]; since memory is reserved in advance without knowing the exact requirements of the thread, a certain amount of memory overhead is likely. On the other hand, the entirety of all threads has to be orchestrated by an operating system module called *scheduler*, which requires processing time relative to the number of threads [40]. Moreover, a computationally expensive procedure called *context switching* must also be followed upon changing the actively processed thread [35]. This process includes complications called *buffer* and *cache misses* as well as lock contentions [28, p. 2]. When a certain number of active threads is reached, this can lead to serious performance degradation, as illustrated in figure 2.4. Especially when the application is executed inside a *virtual machine*¹² – which is often the case for distributed applications due to better replicability – the over-provisioning of memory leads to scarce resources [12, p. 1].

¹¹Certain 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/>).

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



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

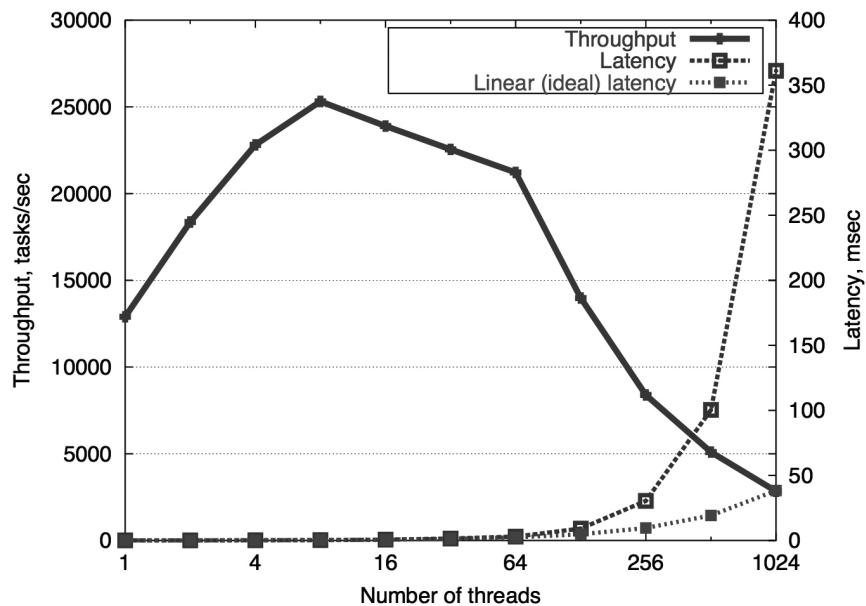


Figure 2.4: This graph shows the performance degradation resulting from rising request frequency for a purely thread-based Web server. Because of the performance overhead introduced by a large number of concurrent threads, the processing throughput decreases. If the number of threads grows very large,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

Some of the problems of threads can be addressed by using a *thread pool*: Instead of spawning new threads upon each request, a fixed number of threads is spawned in advance and workload is distributed among them. However, this procedure is not without problems and introduces the delicate step of setting the thread pool size [32]. It can be concluded that a lower thread overhead can benefit the overall performance of a process. Furthermore, when scaling an application, the maximum number of simultaneously

processed threads can at best only increase linearly in relation to the number of processing cores in a system [34].

2.2.2 Event-based Model

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

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

Relation to Threads

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

Flow of Control

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

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

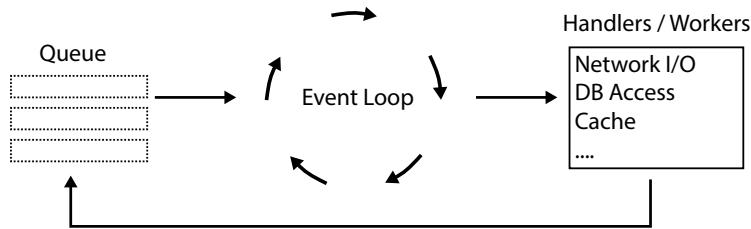


Figure 2.5: Basic flow of control in an event-driven application. Operations that would normally block the event loop are executed separately and create further events upon completion.

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

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

Departure From the Call Stack

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

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

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

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

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

```

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

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

```

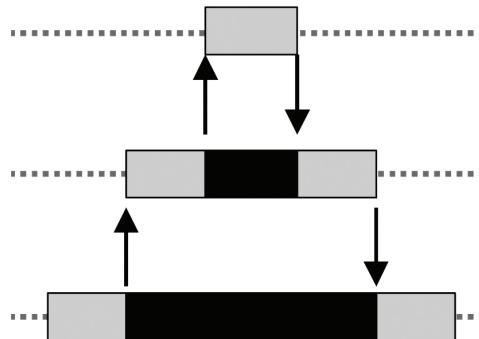


Figure 2.6: Illustration of a simple call stack structure. As time progresses (horizontal axis, left to right), the call stack grows with each function call. As the functions return gradually, the stack size decreases again. Active parts of functions are shown in grey, passive (i.e. *waiting*) parts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

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

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

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

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

Scalability

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

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

To scale a single event loop on one machine, one event loop process can be created for every processing core – on a machine with a quad-core processor, four event loops can process incoming requests in parallel¹¹. In such on-system scaling situations, event-based systems have the advantage, that due to the reduced mutual exclusiveness of the program code, less locking



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

situations occur. An example for this is, when a queue is accessed by multiple threads - only one thread can access the queue simultaneously, the other one has to wait.

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

Drawbacks

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

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

Additionally, event-based programs lack certain compiler¹³ optimisations (given that the programming language used is compiled, rather than inter-

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

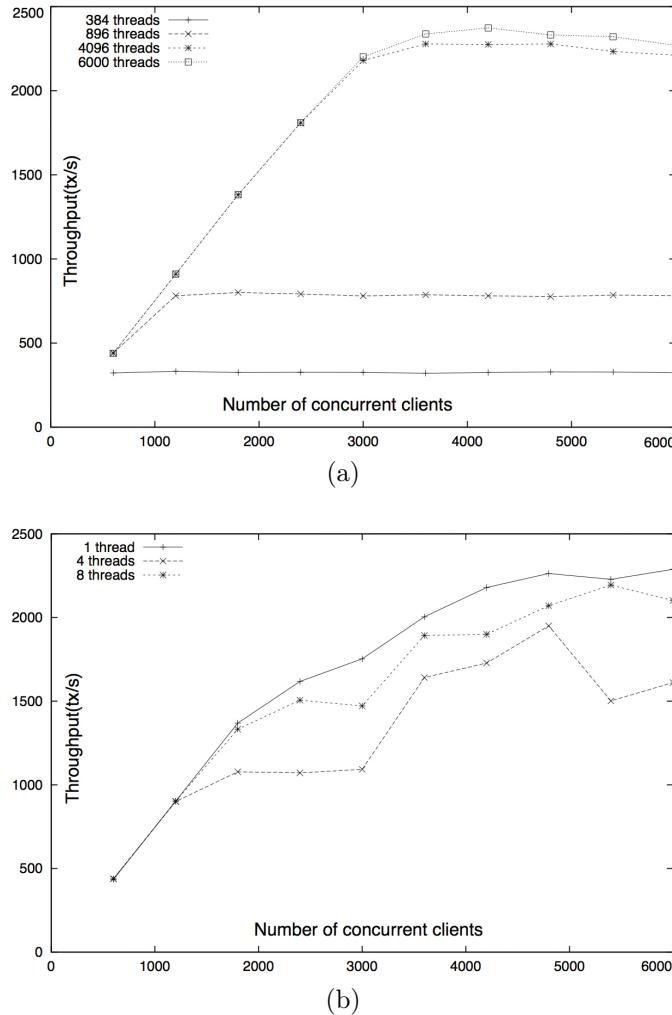


Figure 2.8: Typical scaling behaviour of a purely thread-based application versus an event-based application. This example is taken from an experiment by D. Carrera et al. [5]. The graphs show the effect of an increasing number of clients (x-axis) on the request throughput (y-axis) using a standard *httpd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

interpreted) like advanced memory management, inline functions¹⁴ and compile-time warnings about race conditions [2, p. 5].

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

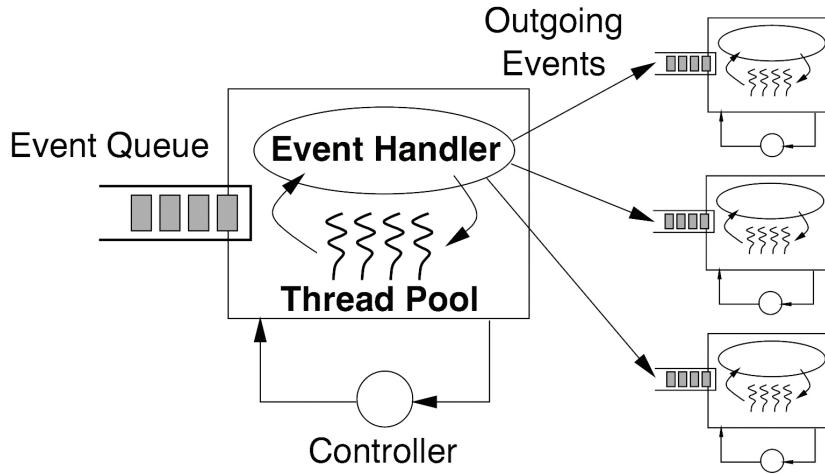


Figure 2.9: Illustration of a single SEDA stage (left) and the communication with other stages (right). Each stage has its own event queue, event handler, thread pool and resource controller. Image source: [28]

2.2.3 Staged Event-driven Architecture

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

Flow of Control

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

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

Drawbacks

The coupling of stages by means of event queues is not necessarily supporting a clear application structure. M. Welsh, one of the developers mainly involved in the invention of SEDA in 1999, states this coupling as a main problem of SEDA [43]:

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 [43]. Apart from that, queues are considered an unsuitable data structure for high-concurrency applications, because they do not allow for concurrent access by multiple parties. Lastly, SEDA requires a fair amount of fine-tuning on the part of the developer in order to function flawlessly [38].

2.2.4 Actor Model

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

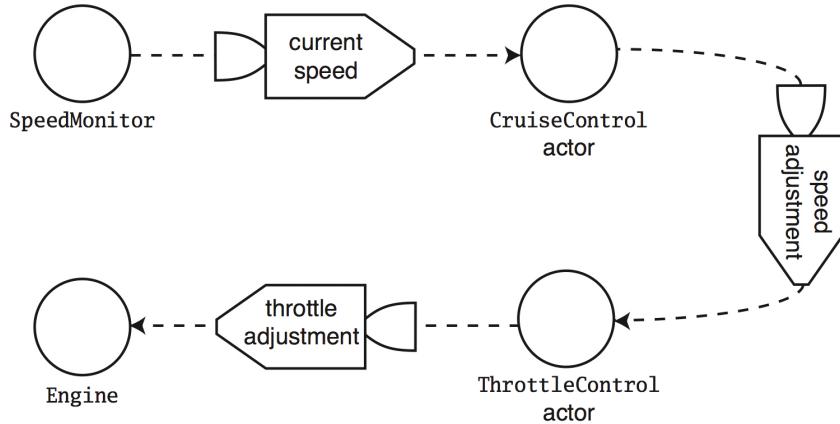


Figure 2.10: Illustration of an actor-based flow of control. Actors are represented as circles and messages are represented as rockets. The speed monitor actor periodically sends a message containing the current speed to the cruise control actor. If a deviation of the desired speed is detected, an adjustment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

Flow of Control

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

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

Incoming messages are handled via *mailboxes*, which are basically actor-specific queues that receive, filter and defer incoming messages. Like in event-based architectures, there is no guarantee in which order messages arrive in the mailbox and – since the actor model does not specify a medium for

passing messages (see section *Scalability*), a message can take arbitrarily long to reach its destination [7, p. 97]. Since only the corresponding actor can access his mailbox and only one message is processed at a time, no concurrency issues (like race conditions) can arise for the actor state and the mailbox as well as in between messages [8, p. 12].

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

Scalability

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

Drawbacks

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

Chapter 3

State of the Art

As already mentioned in section 2.1, *Development*, developing software is greatly facilitated using existing building blocks instead of writing the entirety of program code from scratch. Especially Web server applications benefit from *frameworks* – i.e. third-party software that can be extended with application-specific code – because frameworks generally provide support for standard, repetitive procedures like handling network communication, database access, caching¹ and URL mapping². The process of minimising repetitive code and maximising code reusability can be described as reducing *boilerplate code* or by the slogan *DRY*³ [16, p. 149] [22, p. 1].

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

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

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

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

³Don't repeat yourself

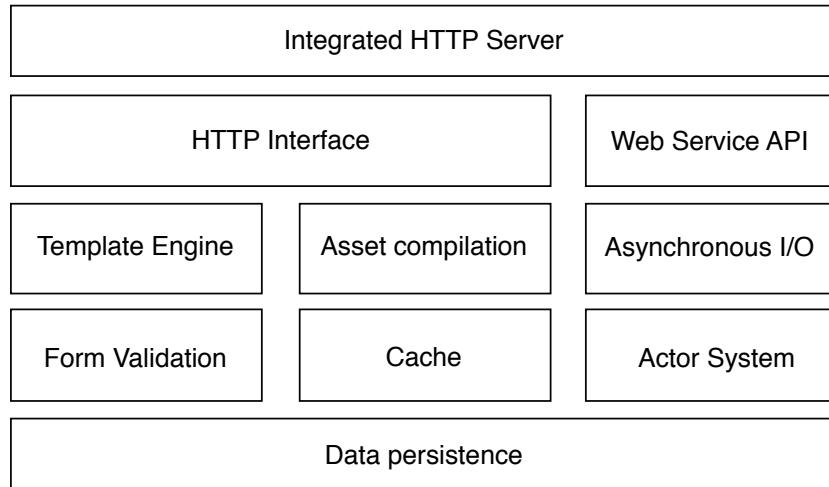


Figure 3.1: A Web framework aims to facilitate development by including frequently used capabilities. Inbound and outbound network communication represents a large share of common features (top three elements). Also, served assets like websites and images play a role in certain use-cases (but do not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

3.1 Event-based technologies

3.1.1 Node.js

*Node.js*⁴ is not only a framework, but rather a dedicated open source software platform for purely event-driven applications. User-level code is written in *JavaScript* and interpreted by the *V8*⁵ engine also used in the *Google Chrome*⁶ Web browser [18, p. 19]. This is reasonable since *JavaScript* was originally used primarily for programming client-side website behaviour; however, *Node.js* uses a module system⁷ to add various Web server-specific features. These features include – among others – networking abstractions, file and operating system abstractions and replication and scheduling utilities⁸. The platform also includes its own Web server via the `http` module.

⁴<http://nodejs.org/>

⁵<https://code.google.com/p/v8/>

⁶<https://www.google.com/intl/en/chrome/browser/>

⁷Using the *CommonJS* module specification (<http://www.commonjs.org/>)

⁸See <http://nodejs.org/api/> for an exhaustive list of system modules or <https://www.npmjs.org/> for a popular extension module repository

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

```

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

```

Development

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

Due to the functional nature of *JavaScript*, callback functions can be the main driving force of asynchronous program flow. The following factors support this [41]:

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

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

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

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

```

1 step1(function (value1) {
2     step2(value1, function (value2) {
3         step3(value2, function (value3) {
4             step4(value3, function (value4) {
5                 // Do something with value4
6             });
7         });
8     });
9 });
10

```

the popular *jQuery* library⁹ or the *Q* library¹⁰, sequential functions can be written more idiomatically as a chain of commands (see program 3.3). When a promise is created, the result is *deferred*, i.e. returned at a later point in time. If the action of the promise was successful, the promise is *resolved*, otherwise it is *rejected*. There is also a comprehension that allows for resolving multiple promises in parallel and treating the results as a single array of values as soon as all promises are resolved:

```

1 Q.all([stepA, stepB]).then(function (results) {
2     var resultA = results[0];
3     var resultB = results[1];
4 });

```

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

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

```

1 async.parallel([
2     function(){ ... },
3     function(){ ... }
4 ], callback);

```

⁹<http://jquery.com/>

¹⁰<https://github.com/krisbowal/q>

¹¹<https://github.com/caolan/async>

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

```

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

```

Program 3.4: A simple example of explicit events. First, the emitter created through the `events` module registers a behaviour (in form of a callback function) for a certain event type (i.e. `doorOpen`). At an arbitrary point in time an event of this type is created and triggers the callback function. Code source: [29]

```

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

```

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

Node.js applications can also benefit from certain framework modules that add *MVC* (Model-View-Controller) capabilities. One such module is the *express* framework¹², which includes features such as advanced routing and templates and brings *Node.js* one step closer to being a full-stack Web

¹²<http://expressjs.com/>

framework (see section 3).

Scalability

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

Applications running on *Node.js* per default only use a single thread for processing [41]. As mentioned in the previous chapter, this has a positive effect on scheduling overhead. Because of the nature of *JavaScript* and *Node.js* (e.g. asynchronous networking and file abstractions, use of callbacks), it is comparably easy to write code that does not block the processing thread. However, *if* blocking occurs, the consequence is that the whole application is unable to process any requests until the blocking action has finished. On the other hand, this removes any need for synchronisation concerns and prevents address space conflicts between threads [7, p. 105].

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

Performance

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

3.1.2 Eventmachine

Ruby¹³ is a dynamic programming language that has a high adoption rate due to the popular *Ruby on Rails* MVC framework that powers a lot of

¹³<https://www.ruby-lang.org>

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

```

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

```

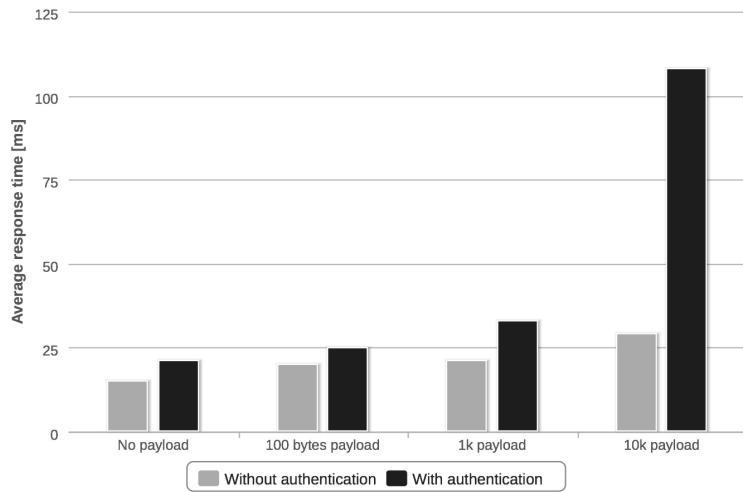


Figure 3.2: In this figure taken from a performance analysis of D. Torstenson and E. Elof, requests are sent to a *Node.js* server with different payload sizes. The requests are sent both with and without authentication; authentication is done by hashing (i.e. processing) the whole payload using the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

Program 3.6: This program demonstrates how blocks can be used with the `TCPServer` library (included in *Ruby's* standard library) to create a new thread for every incoming network client. The block (line 4 to 8) acts as a container applied to the result of previous operations, similar to a closure in *JavaScript*; the `client` variable is the result of the `new` method of the `Thread` class, which accepts a `TCPSocket` object. Code source: [33]

```

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

```

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

Development

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

To create a non-blocking networking server in *Ruby*, more complex operations are needed. This includes creating an event-loop¹⁴ and using the `IOL` class and the `accept_nonblock` method of the `TCPSocket` class to create logical concurrency on one thread. Operations like handling connections and reading input from the sockets also have to be managed by the developer [33].

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 *Connections*, which have to be defined within the reactor loop. *EventMachine*

¹⁴Strictly speaking, since no events are involved this is called a *reactor loop* [33].

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

```

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).

Scalability and Performance

3.1.3 Others

React, Derby, Twisted, Java NIO

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.

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

3.2 Actor-based technologies

3.2.1 Play!

a

a

a

a

a
The Scala Actor Model

3.2.2 Lattice

a

a
vs rails

3.2.3 Others

Lift

Chapter 4

Implementation

PIXXERS Bla Bla Bla

4.1 Project Setup

4. Implementation

40

a Just an example, no Real appPlay Setup, Threadpools, Local Testing, Cloud Deployment

4.2 Development

a

a

a

4.3 Deployment

a

a

4.4 Maintenance

a

a

Chapter 5

Evaluation

5.1 Approach

a

5. Evaluation

50

a

a
Methodology

5.2 Performance

a

a

a

a JMeter loader.io

5.3 Maintenance

a

a

5.4 Results

a

a

Chapter 6

Conclusion and Future Development

6. Conclusion and Future Development

61

a

6. Conclusion and Future Development

62

a

References

Literature

- [1] J Armstrong. *Programming Erlang: Software for a Concurrent World*. 2007. URL: <http://dl.acm.org/citation.cfm?id=1403889>.
- [2] Rob Von Behren, Jeremy Condit, and Eric Brewer. “Why Events Are A Bad Idea (for high-concurrency servers)”. In: *HotOS IX : The 9th Workshop on Hot Topics in Operating Systems* (2003).
- [3] Jonas Bonér. “The Reactive Manifesto”. 2013.
- [4] Daniele Bonetta, Danilo Ansaloni, and Achille Peternier. “Node.Scala: Implicit Parallel Programming for High-Performance Web Services”. In: *Lecture Notes in Computer Science Volume 7484* (2012).
- [5] David Carrera et al. “Evaluating the Scalability of Java Event-Driven Web Servers”. In: *ICCP 2004* (2004).
- [6] Vaarnan Drolia, Cedric Ansley, and Chin Shen. “Threads vs Events for Server Architectures”. 2010.
- [7] Benjamin Erb. “Concurrent Programming for Scalable Web Architectures”. PhD thesis. University Ulm, 2012.
- [8] Joakim Eriksson. “Representation of asynchronous communication protocols in Scala and Akka”. PhD thesis. Linköpings Universitet, 2013.
- [9] Jeffrey Fischer, R Majumdar, and Todd Millstein. “Tasks: language support for event-driven programming”. In: *PEPM '07 Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (2007), pp. 134–143.
- [10] Michael Fitzgerald. *Learning Ruby*. O'Reilly Media, Inc, 2007. URL: http://books.google.com/books?hl=en\&lr=\&id=pYS__Fm5LqUYC\&oi=fnd\&pg=PR11\&dq=Learning+Ruby\&ots=YmKGFWTaj\&sig=KQ5BeDI83fTDqqXppkoeumgtnyM.
- [11] Munish K. Gupta. *Akka Essentials*. Packt Publishing, 2012.
- [12] Philipp Haller and Martin Odersky. “Event-Based Programming without Inversion of Control”. In: *Modular Programming Languages* (2006). URL: http://link.springer.com/chapter/10.1007/11860990__2.

- [13] Philipp Haller and Martin Odersky. "Scala Actors: Unifying Thread-based and Event-based Programming". In: *Theoretical Computer Science* 410.2-3 (Feb. 2009), pp. 202–220. URL: <http://linkinghub.elsevier.com/retrieve/pii/S0304397508006695>.
- [14] Philipp Haller and Frank Sommers. *Actors in Scala*. 1st ed. 2011. URL: <http://scholar.google.com/scholar?hl=en\&btnG=Search\&q=intitle:Scala\#0> <http://dl.acm.org/citation.cfm?id=2222540>.
- [15] Cal Henderson. *Building Scalable Web Sites*. O'Reilly Series May. O'Reilly Media, Inc, 2006, p. 330.
- [16] Peter Hilton, Erik Bakker, and Francisco Canedo. *Play for Scala: Covers Play 2*. 1st ed. Manning Publications Co., 2013, p. 300.
- [17] Gregor Hohpe. "Programming Without a Call Stack à Event-driven Architectures". In: *Enterprise Integration Patterns* (2006).
- [18] Tom Hughes-Croucher and Mike Wilson. *Node - Up and Running*. 2012.
- [19] Edward A. Lee. "The problem with threads". In: *Computer* 39.5 (May 2006), pp. 33–42.
- [20] Paul Mackay. "Why Has the Actor Model Not Succeeded?" In: *surprise 97* (1997).
- [21] S Nadimpalli and S Majumdar. "Techniques for Achieving High Performance Web Servers". In: *Parallel Processing, 2000. . .* (2000).
- [22] R Orsini. *Rails Cookbook*. O'Reilly Media, Inc, 2008.
- [23] Alexander Reelsen. *Play Framework Cookbook*. Packt Publishing, 2011.
- [24] Sencha Inc. "Web Applications Come of Age". 2011.
- [25] Stefan Tilkov and Steve Vinoski Verivue. "Node.js : Using JavaScript to Build High-Performance Network Programs". In: *IEEE The Functional Web* (2010).
- [26] Daniel Torstensson and Erik Elof. "An Investigation into the Applicability of Node. js as a Platform for Web Services". PhD thesis. 2012.
- [27] Bryan Veal and Annie Foong. "Performance Scalability of a Multi-Core Web Server". In: *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems - ANCS '07* (2007), p. 57.
- [28] Matt Welsh, David Culler, and Eric Brewer. "SEDA : An Architecture for Well-Conditioned , Scalable Internet Services". In: *SOSP-18* (2001).

Online Sources

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

- [30] R Fielding, U C Irvine, and J Gettys. *Hypertext Transfer Protocol – HTTP/1.1*. 1999. URL: <http://www.ietf.org/rfc/rfc2616.txt>.
- [31] Jesse James Garrett. *AJAX : A New Approach to Web Applications*. 2005. URL: <http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/>.
- [32] Brian Goetz. *Thread Pools and Work Queues*. 2002. URL: <http://www.ibm.com/developerworks/java/library/j-jtp0730/index.html>.
- [33] Aman Gupta. *EventMachine*. 2010. URL: https://dl.dropboxusercontent.com/u/635/em_export.pdf.
- [34] Michael McCool. *The Serious Drawbacks of Explicit Multi- Threading*. 2008. URL: <http://software.intel.com/en-us/blogs/2008/06/05/nitrogen-narcosis-part-ii-the-serious-drawbacks-of-explicit-multi-threading/>.
- [35] Mark McGranaghan. *Threaded vs Evented Servers*. 2010. URL: <http://mmcgrana.github.io/2010/07/threaded-vs-evented-servers.html>.
- [36] Svetlin Nakov. *Internet Programming with Java*. 2004. URL: <http://www.nakov.com/inetjava/lectures/part-1-sockets/InetJava-1.3-Multithreading.html>.
- [37] Oracle. *Establishing Performance Goals*. 2010. URL: <http://docs.oracle.com/cd/E19900-01/819-4741/fygaj/index.html>.
- [38] Michael Peterson. *Events and Event-Driven Architecture : Part 1*. 2012. URL: <http://thornydev.blogspot.co.at/2012/01/events-and-event-driven-architecture.html>.
- [39] Pingdom. *A History of the Dynamic Web*. 2007. URL: <http://royal.pingdom.com/2007/12/07/a-history-of-the-dynamic-web/>.
- [40] Mark Russinovich. *Pushing the Limits of Windows : Processes and Threads*. 2009. URL: <http://blogs.technet.com/b/markrussinovich/archive/2009/07/08/3261309.aspx>.
- [41] Mikito Takada. *Understanding the Node.js Event Loop*. 2011. URL: <http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/>.
- [42] Webopedia. *Multi-core Technology*. 2007. URL: http://www.webopedia.com/TERM/M/multi_core_technology.html.
- [43] Matt Welsh. *A Retrospective on SEDA*. 2010. URL: <http://matt-welsh.blogspot.co.at/2010/07/retrospective-on-seda.html>.