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

Felix Hessenberger

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2014

Declaration

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

Hagenberg, June 30, 2014

Felix Hessenberger

Contents

Abstract

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

Kurzfassung

Franz rast im komplett verwahrlosten Taxi quer durch Bayern.

Chapter 1

Introduction

1.1 Motivation

a

a

1.2 Objective

a

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

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 [20]. To minimise networking latency, it is preferable for a Web server to have a high-speed connection to the Internet, fast system I/O[2] and capable routing hardware. However, these parameters are not directly related to software and are thus neglected during the further course of this thesis.

2.1.2 Dynamic Content

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

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

content accordingly [20]. 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 [28]. 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 [28].

2.1.3 Asynchronous Requests

The advent of *AJAX*[6] and mobile applications in the late 2000's changed requirements drastically. Rather than refreshing the whole view for every piece of information sent and received, data could now be transferred in a more granular fashion. By asynchronously communicating with an API[7] endpoint in the background, operations like deleting an item from a list could be performed invisibly and ubiquitously without reloading the page context. Especially applications that aim to provide desktop-like capabilities – commonly called Rich Internet Applications – make heavy use of asynchronous requests [16, p. 4]. This inherently also changed users' expectations for websites from anticipating a certain amount of load time to implicating real-time behaviour [21]. 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 [14, 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 [26]. At this point, the server may chose to neglect the request (ideally by returning the status code *503 Service Unavailable* [20]), not respond at all or even stop serving clients altogether (i.e. "crash").

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

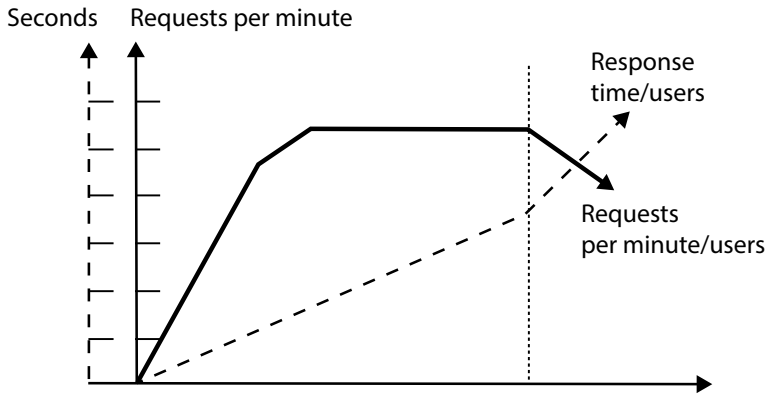

Figure 2.1: Correlation between request frequency and response time in a typical Web server setup. After the server has reached its limit of linearly serving clients (indicated by the dotted line), response times become inversely proportional to the request frequency. Image source: [26]

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 [4, 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 [12, p. 11].

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[8], scaling out on more than one machine will be unsuccessful [18]. Scalability can be measured by the relationship between hardware resources and the increase of performance. If this relationship is nearly linear, the system can be considered to scale well.

[8]Random Access Memory

2.1.6 Development

Not a part of the production system itself, but nonetheless an essential part of all Web server applications is their development. A structured, idiomatic way of writing application logic doubtlessly contributes to every software product. Modularisation of components facilitate the use of third-party software like libraries and frameworks. In return, using existing solutions can greatly reduce development time and effort, while simultaneously providing proven solutions. Web server applications particularly benefit from frameworks since they often handle standard tasks like network I/O^2, database access and caching [15, 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 [4, 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 [30]. This section lists various paradigms associated with designing an application capable of maintaining multiple flows of control.

2.2.1 Purely Thread-based

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 [25].

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

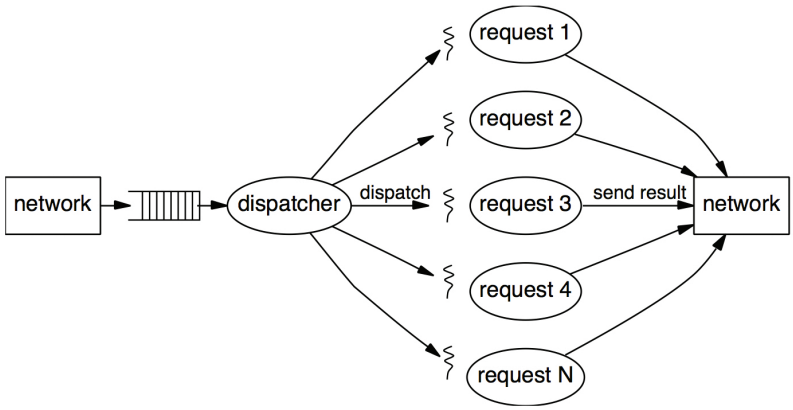

Figure 2.2: On a purely thread-based Web server, each request is handled by a dedicated thread. Incoming network requests are queued and sequentially accepted by a dispatcher, that distributes them among available threads – either by using idle threads from a thread pool or by creating new threads. Image source: [19]

communication is commonly treated as a set of mutually independent connections; this approach of abstraction facilitates a clear program flow structure [18]. 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 [1, 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 [7, p. 1]. Therefore, the implementation of large-scale systems heavily relying on threads always introduces additional complexity [13, 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 [10, p. 162]. This behaviour can be observed for instance in implementations of the popular LAMP[9] server stack configuration [10, p. 48]. A less experienced programmer might find this ideal, since concurrency stays mostly 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

[9] Linux, Apache, MySQL, PHP

processing, i.e. physical concurrency, is limited by the number of processing cores. This means that on a computer equipped with a quad-core processor, four threads can be executed – and thus, four requests can be served – in parallel[10].

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 [10, p. 196]. When a thread encounters a blocking operation, it cannot advance further in the program flow until the operation completes, as illustrated in figure 2.3. The resulting delay can account to anywhere from a few milliseconds to several seconds, for instance when accessing a slow or irresponsive Web service. The only way to counteract the temporary occupation of threads and to continue processing incoming requests is the creation of new threads [12, 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 [29]; 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 [29]. Moreover, a computationally expensive procedure called *context switching* must also be followed upon changing the actively processed thread [24]. This process includes complications called *buffer* and *cache misses* as well as lock contentions [19, 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*[11] – which is often the case for distributed applications due to better replicability – the over-provisioning of memory leads to scarce resources [8, 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

[10] 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/).

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

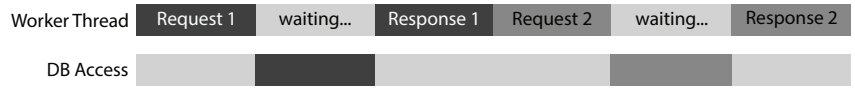

Figure 2.3: A typical blocking situation in Web server scenario. When request one (shown in dark grey) arrives, a database operation is necessary. During the course of this operation, the executing thread blocks while waiting for results. The response can only be sent when data is returned and the next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

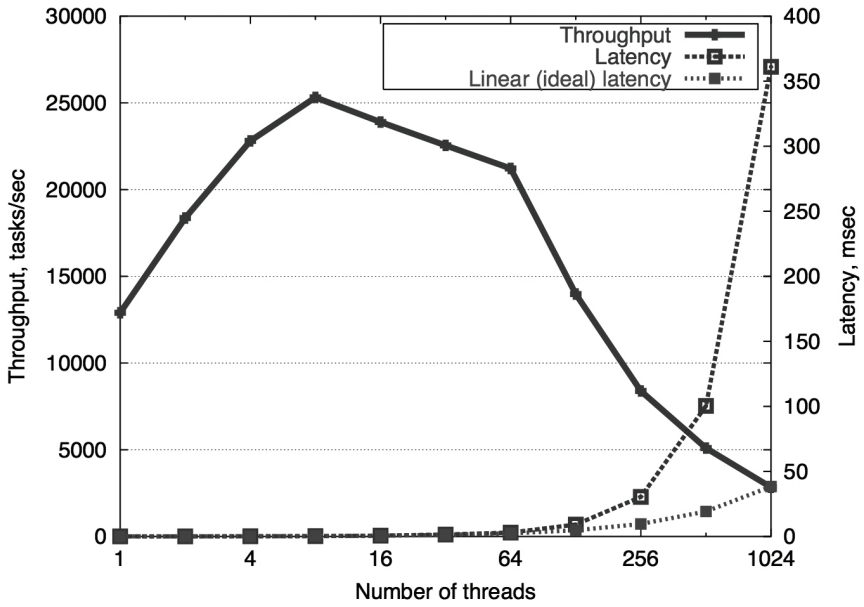

Figure 2.4: This graph shows the performance degradation resulting from rising request frequency for a purely thread-based Web server. Because of the performance overhead introduced by a large number of concurrent threads, the processing throughput decreases. If the number of threads grows very large, the request-response latency escalates due to the shortage of resources. The data is taken from an experiment by M. Welch et al. [19]. Image source: [19]

step of setting the thread pool size [22]. 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 [23].

2.2.2 Event-based

While threads on their own present a convenient abstraction for handling Web requests, recent years have seen an incline towards event-driven architectures [17]. 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* [11].

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

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 and handlers [12, 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

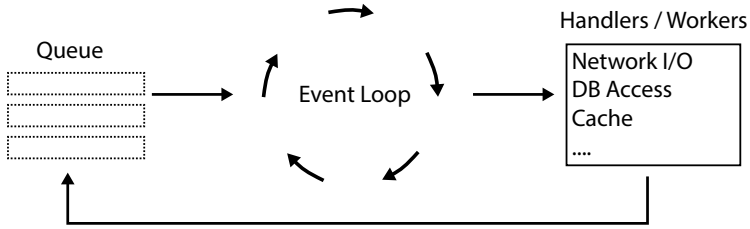

Figure 2.5: Basic flow of control in an event-driven application. Operations that would normally block the event loop are executed separately and create further events upon completion.

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

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

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

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 [5, p. 92]. Callback functions are often used to handle results of a blocking operation – like making a request to a remote Web service – directly upon its completion and thus maintaining control over the execution order of further actions. Implementation is typically done via either a function pointer or an *annonymous function*, as demonstrated in program 2.1.

Departure From the Call Stack

Neither events nor callback functions are usually found in traditional – i.e. sequential – programming. In nearly all application environments the default way of executing routines in succession is to use functions, whose return values are used in the further course of the program. This proven concept

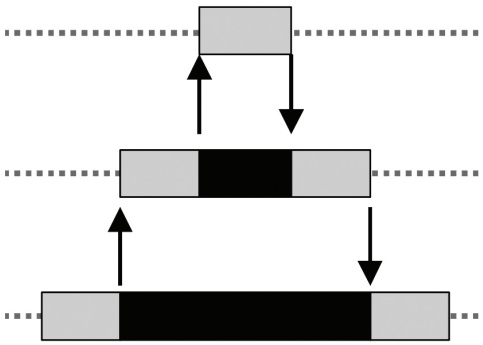

Figure 2.6: Illustration of a simple call stack structure. As time progresses (horizontal axis, left to right), the call stack grows with each function call. As the functions return gradually, the stack size decreases again. Active parts of functions are shown in grey, passive (i.e. *waiting*) parts are shown in black. Image source: [11]

determines three major aspects of program flow [11, 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

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

Despite its advantages, heavy use of the stack in the context of modern Web applications poses two substantial problems: On the one hand, the concept of the stack origins from a time where concurrent computing was not frequently used – especially in the manner seen in highly concurrent applications. The behaviour of the stack pursues a strongly linear manner of execution. Because at any point in time, only one call can be pushed onto the stack at once, only one action can happen at a time. Likewise, if an operation were to take an extended or unspecified period of time – like accessing a remote Web service – a single call stack provides no way of executing another operation during this time. The second problem of the call stack is that it can not be distributed across physically or logically separated

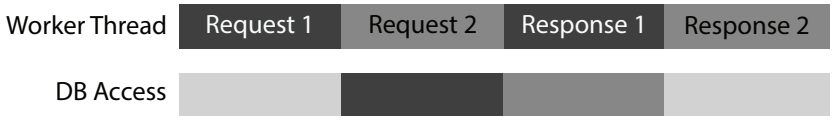

Figure 2.7: A non-blocking situation in Web server scenario. When request one (shown in dark grey) arrives, a database operation is necessary. This is a blocking operation, but since the worker thread does not have to wait for it to complete, the next request (shown in medium grey) can already be accepted 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.

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 [11].

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* [4].

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 [3, 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 [13].

To scale a single event loop on one machine, one event loop process can

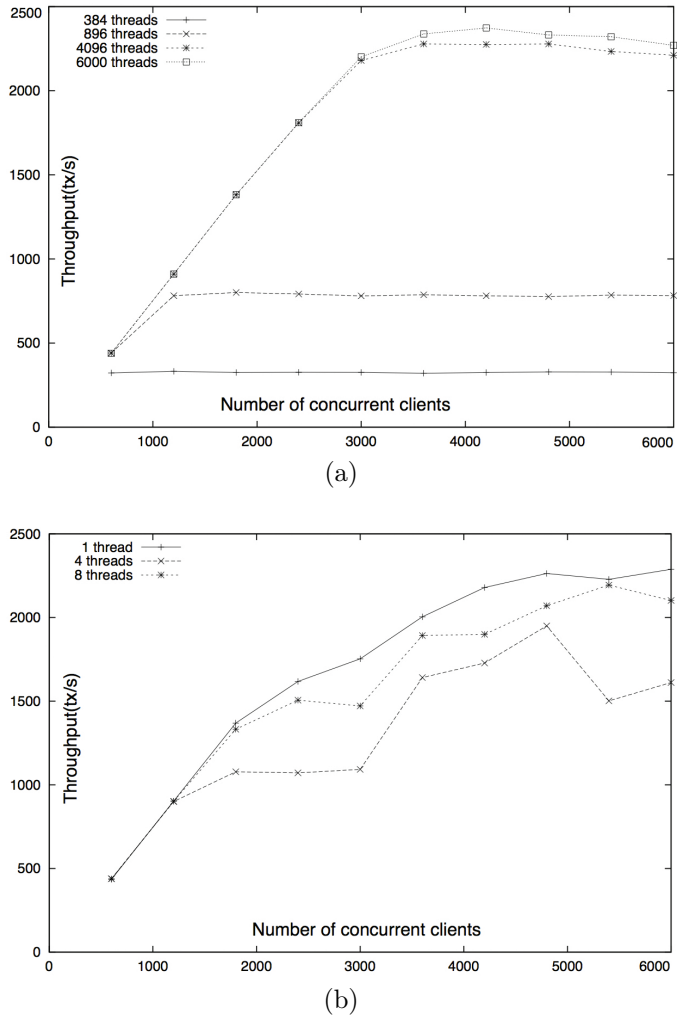

Figure 2.8: Typical scaling behaviour of a purely thread-based application versus an event-based application. This example is taken from an experiment by D. Carrera et al. [3]. The graphs show the effect of an increasing number of clients (x-axis) on the request throughput (y-axis) using a standard *httdp 2.0* Web server (http://httpd.apache.org/) – shown in (a) – and using *Java*'s event-based *NIO* interface (http://www.oracle.com/) – shown in (b). Image source: [3].

be created for every processing core – on a machine with a quad-core processor, four event loops can process incoming requests in parallel[10]. 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 situations occur. An example for this is, when a queue is accessed by multiple threads - only one thread can access the queue simultaneously, the other one

has to wait.

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

Drawbacks

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

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

Additionally, event-based programs lack certain compiler[12] optimisations (given that the programming language used is compiled, rather than interpreted) like advanced memory management, inline functions[13] and compile-time warnings about race conditions [1, p. 5].

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

[12] 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.

[13] 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.

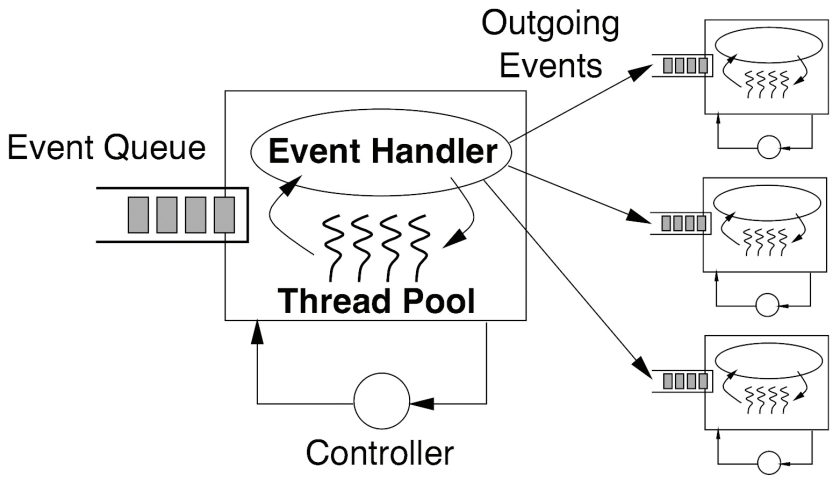

Figure 2.9: Illustration of a single SEDA stage (left) and the communication with other stages (right). Each stage has its own event queue, event handler, thread pool and resource controller. Image source: [19]

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 [19].

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 protection. An application that balances its own resources and makes decisions based on parameters like demand and response time is called a *conditioned* application [19].

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 [31]:

> 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 [31]. 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 [27].

2.2.4 The 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 [8].

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 [8]. 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.

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

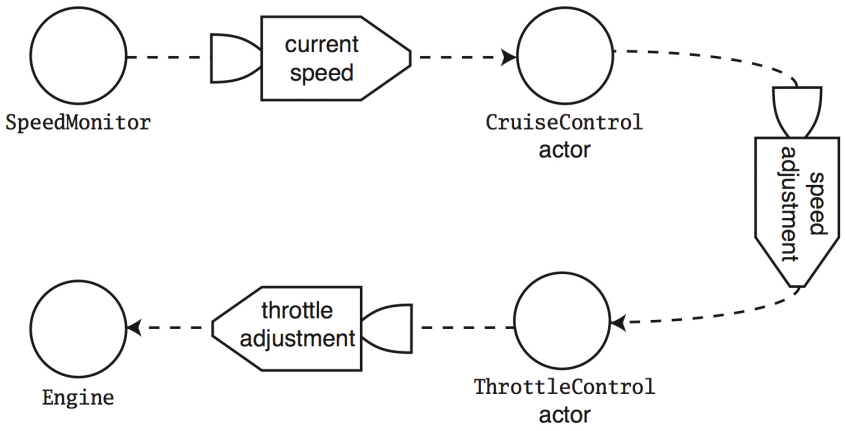

Figure 2.10: Illustration of an actor-based flow of control. Actors are represented as circles and messages are represented as rockets. The speed monitor actor periodically sends a message containing the current speed to the cruise control actor. If a deviation of the desired speed is detected, an adjustment message is sent to the throttle control actor, who again sends a message to the engine. Image source: [9]

introduces symmetry to logical communication. Furthermore, an actor's internal state can only change in response to incoming messages [9, p. 38]. This simple fact has great implications on state management across the application; this architectural characteristic is known as *share-nothing* architecture [2, 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 [6, p. 12].

Scalability

actors are lightweight resplicatoin

Drawbacks

2.2.5 Reactive Architecture

Chapter 3

State of the Art

3.1 Event-based

3.1.1 Express

a

a

3.1.2 Watson

a

3.1.3 Twisted

a

3.1.4 Node.js

a

3.2 Actor-based

3.2.1 The Scala Actor Model

3.2.2 Spray

a

a

3.2.3 Lift

a

a

3.2.4 Play! Framework

a

a

a

a

a

3.3 Other

3.3.1 Ruby on Rails

a

a

3.3.2 Node.scala

a

a

a

a

Chapter 4

Implementation

4.1 Considerations

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

4.2 Synchronous Version

a

a

a Prerequisites, ...

4.3 Asynchronous Version

a

a

a

Chapter 5

Evaluation

5.1 Approach

a

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

a

a

References

Literature

[1] 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).

[2] 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).

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

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

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

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

[7] 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.

[8] 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.

[9] 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.

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

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

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

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

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

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

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

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

[18] 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.

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

Online Sources

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

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

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

[23] 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/.

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

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

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

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

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

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

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

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