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

1.3 Structure

Chapter 2

Technical Background

2.1 Terms and Definitions

A Web server can be utilised to handle rather different tasks, from serving entire Web pages to representing an endpoint for raw data retrieval to merely delivering static assets like images. 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 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 [13]. 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 requests and respond with static content

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

accordingly [13]. With the release of *PHP*[3], *ASP*[4] and *Java*[5] – 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 [21]. 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 [21].

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 [11, p. 4]. This inherently also changed users' expectations for websites from anticipating a certain amount of load time to implicating real-time behaviour [14]. 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 [9, 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 [19]. At this point, the server may chose to neglect the request (ideally by returning the status code *503 Service Unavailable* [13]), 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

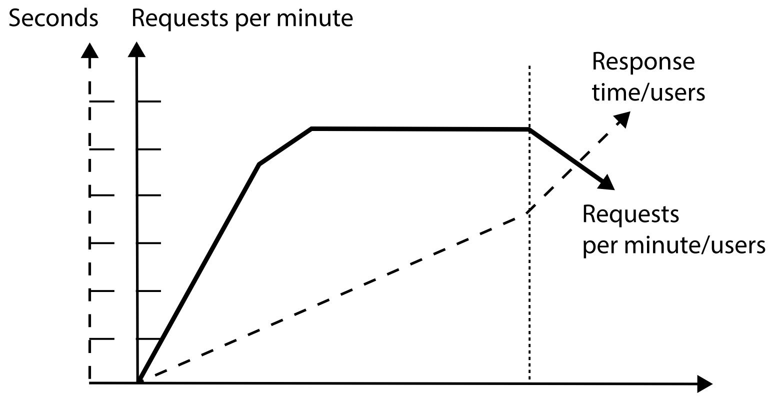

Figure 2.1: Correlation between request frequency and response time in a typical Web server setup. After the server has reached its limit of linearly serving clients (indicated by the dotted line), response times become inversely proportional to the request frequency. Image source: [19]

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 [2, 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 [7, 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 [12]. 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 [10, 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 Patterns

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 [2, 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 with multiple program flows that can be executed concurrently [23]. This section lists various paradigms associated with designing an application capable of maintaining multiple flows of control.

2.2.1 Threads

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 at roughly the same time, thus giving the impression of simultaneous execution. Physical concurrency is inherently more efficient [18].

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 [12]. Accordingly, every request can be treated as an isolated flow of control [1, p. 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 database components are accessed sequentially. Thus, close attention has to be paid by the developer to avoid race-conditions, deadlocks and access violations – complications that generally result from improper thread coordination [4, p. 1]. Therefore, the implementation of large-scale systems heavily relying on threads – as an evolutionary improvement from sequential computing – always introduces additional complexity [8, 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 [5, p. 162]. This behaviour can be observed for instance in implementations of the popular LAMP[9] server stack configuration [5, 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 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].

Problems

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 [5, p. 196]. When a thread encounters a blocking operation, it cannot advance in the program flow until the operation completes, as illustrated in figure 2.2. 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

[9]Linux, Apache, MySQL, PHP

[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/).

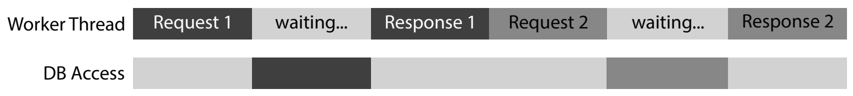

Figure 2.2: A typical blocking situation in Web server scenario. When request one (shown in dark grey) arrives, a database operation is necessary. During the course of this operation, the executing thread blocks while waiting for results. The response can only be sent when data is returned and the next request (shown in medium grey) can only be served after the first one completes.

of new threads [7, 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 [22]; 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 [22]. Moreover, a computationally expensive procedure called *context switching* must also be followed upon changing the actively processed thread [17].

Some of the problems of threads can be addressed by using a *thread pool*: Instead of spawning new threads upon each request, a fixed number of threads is spawned in advanced and workload is distributed among them. However, this procedure is not without problems and introduces the delicate step of setting the thread pool size [15]. 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 [16].

2.2.2 Events

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[11] 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. In the case of an event-driven HTTP server, the returned event may for instance contain an HTTP response to be sent to the initial client. Larger applica-

[11]A queue can theoretically be absent, but this would prevent events from being received while the event loop blocks.

Program 2.1: Calling a callback function via a function pointer in JavaScript.

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

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

```

tions typically have one event loop per process and a number of listeners and handlers [7, p. 33]. For an illustration of a basic event-driven application setup, see figure 2.3.

Flow of Control

While threads present a natural abstraction for handling Web requests, recent years have seen an incline towards event-driven flow of control. Seen from a different perspective, events are at least equally idiomatic: 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* [6].

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 (see section 2.2.3 for the counterpart – staged events). 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 [3, p. 92]. Callback functions are often used to handle results of a blocking operation – like calling a Web service – directly after 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 and 2.2, respectively.

Instead of keeping the entirety of method calls leading to a certain behaviour in a section of reserved memory called the *stack*, event-driven programming makes use of *callback functions*.

Program 2.2: Calling a callback function via an annonymous function in JavaScript.

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

```

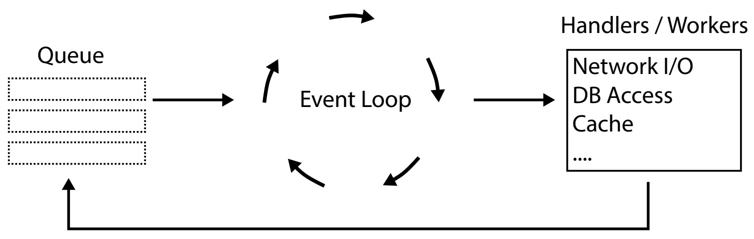

Figure 2.3: Basic flow of control in an event-driven application. Operations that would normally block the event loop are executed separately and create further events upon completion.

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 needs at least one thread. However, this is not a favourable scenario, especially for a Web server: If the event loop and the handler would run on the same thread, the event loop would block if the handler blocks and thus would not be able to accept further events. Therefore the event loop (and with it the application I/O[2]) commonly runs on a dedicated thread with the handlers run on other threads or – in more sophisticated implementations – in a thread pool (see section 2.2.1).

2.2.3 SEDA

In this thesis, the term *event-driven* always describes intra-system event architecture, also known as *staged event-driven architecture* (*SEDA*) [20].

When incoming demand exceeds system processing capacity, it can simply queue up requests in its internal buffer queues.

2.2.4 Actors

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

3.2 Actor-based

3.2.1 Spray

a

a

3.2.2 Lift

a

a

3.2.3 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] Vaarnan Drolia, Cedric Ansley, and Chin Shen. “Threads vs Events for Server Architectures”. 2010.

[3] Benjamin Erb. “Concurrent Programming for Scalable Web Architectures”. PhD thesis. University Ulm, 2012.

[4] 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.

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

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

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

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

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

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

[11] Sencha Inc. “Web Applications Come of Age”. 2011.

[12] 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.

Online Sources

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

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

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

[16] 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/.

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

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

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

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

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

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

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