Concurrent Programming for Scalable Web Architectures

Diploma Thesis
VS-D01-2012

Institute of Distributed Systems
Faculty of Engineering and Computer Science
Ulm University

Benjamin Erb

April 20, 2012

This thesis has been written by Benjamin Erb in 2011/2012 as a requirement for the completion of the diploma course Media Informatics at Ulm University. It has been submitted on April 20, 2012.

Benjamin Erb
Mail: mail@benjamin-erb.de
WWW: http://www.benjamin-erb.de

Institute of Distributed Systems
Faculty of Engineering and Computer Science, Ulm University
James-Franck-Ring
89081 Ulm, Germany

Title Image

Bustling Beijing by Trey Ratcliff, released under CC-BY-NC-SA 2.0 license.
http://www.flickr.com/photos/stuckincustoms/5069047950/
http://creativecommons.org/licenses/by-nc-sa/2.0/

Icon Sets

Iconic (http://somerandomdude.com/work/iconic/) by P.J. Onori, released under CC-BY-SA 3.0 license.
http://creativecommons.org/licenses/by-sa/3.0/)

Picol (http://picol.org) by Melih Bilgil, released under released under CC-BY-SA 3.0 license.
http://creativecommons.org/licenses/by-sa/3.0/

Abstract

Web architectures are an important asset for various large-scale web applications, such as social networks or e-commerce sites. Being able to handle huge numbers of users concurrently is essential, thus scalability is one of the most important features of these architectures. Multi-core processors, highly distributed backend architectures and new web technologies force us to reconsider approaches for concurrent programming in order to implement web applications and fulfil scalability demands. While focusing on different stages of scalable web architectures, we provide a survey of competing concurrency approaches and point to their adequate usages.

Preface

About this Work

My name is Benjamin Erb and I have been studying Media Informatics at Ulm University since 2006. This work represents a major requirement for the completion of my diploma course in 2012. For a long time, I have been taking a great interest in web technologies, scalable architectures, distributed systems and programming concurrency.

As a consequence, the topic of my thesis covers most of these interests. In fact, it considers the overlap of these subjects when it comes to the design, implementation and programming of scalable web architectures. I hope to provide a comprehensive introduction to this topic, which I have missed so far. As such a primer might also be interesting for many others, I am happy to release my thesis under a free license to the general public.

This thesis incorporates both academic research papers and practically orientated publications and resources. In essence, I aimed for a survey of different approaches from a conceptual and theoretical perspective. Hence, a quantitative benchmarking of concrete technologies was out of scope for my work. Also, the extents of the subjects brought up only allowed for a brief overview. The bibliography and the referenced resources provide a good starting point for further readings.

I am very interested in your feedback, your thoughts on the topic and your ideas! Feel free to contact me (http://www.benjamin-erb.de) or get in touch with me via Twitter: @b_erb

Acknowledgements

First of all, I would like to thank my advisor *Jörg Domaschka* for his steady and precious support. He gave me substantial liberties, but was always available for a lot of good advices and formative chit-chats when I needed them.

I also thank my supervisors, *Prof. Dr. Hauck* and *Prof. Dr. Weber*, especially for allowing me to work on such a conceptual and comprehensive topic for my thesis. After all, both of them need to be held responsible for my interests in these topics to some degree.

I want to thank all of my proofreaders—namely and alphabetically— *Christian Koch*, *Katja Rogers*, *Linda Ummenhofer*, *Lisa Adams*, *Matthias Matousek*, *Michael Müller*, *Nicolai Waniek* and *Timo Müller*. I also want to thank *Jan Lehnardt* and *Lena Herrmann* who reinforced my decision to release this thesis.

Finally, I want to say to everyone, who—directly or indirectly—helped me and supported me during the time I wrote this thesis: *Thank you!*

A labyrinth of symbols… An invisible labyrinth of time.

— Jorge Luis Borges, The Garden of Forking Paths

Contents

1 Introduction

1.1 Motivation

Computers have been increasingly influencing our lives in the last decades. Distinct eras of computing have elapsed, driven by technological progress and affecting the way we are using computers. These shifts of paradigms were motivated mostly by advances in hardware and software technology, but also by changes in the way humans interact with computers.

The first computers were large machines solely built for dedicated applications such as numeric computations. There was no real distinction between software and hardware, and new applications could not be executed without changing the physical configuration. The instructions were executed directly without any operating systems underneath and the program had direct access to all system resources. Also, there was no interaction between the running program and the user. With the advent of punch cards and batch processing, computers became more flexible tools for processing jobs. Not just data—executable programs, too—were now used as input, defined in low-level, imperative languages. There was still no interaction during execution and jobs could only be executed sequentially.

Machines were expensive but in great demand. The next innovations were thus influenced by concepts allowing multiple users to work on the same machine and multiple programs to run at the same time—mainframe computers, operating systems, and time-sharing. Terminal-based command line interfaces provided the first interactive systems. New programming languages such as FORTRAN or ALGOL allowed the development of larger and more reliable applications without using pure assembly code.

The arrival of networking was also the beginning of a new kind of computing. Distributed applications not just take advantage of the resources of multiple machines, they also allow the existence of federated systems that consist of multiple machines remotely located. It promptly became apparent that in order to support such systems, additional software on top of a network operating system was necessary. Others favored a distributed operating system which provided a high degree of transparency and supported migration functions as part of the operating system. Early middleware systems based on transaction monitors and remote procedure calls emerged, easing the development of distributed applications. Existing networks were linked and a global network was formed; the Internet. The evolution from mainframes to mini computers, then workstations

and personal computers, not just urged networking. It also introduced new user interaction mechanisms. Consoles were replaced by graphical displays and enabled more sophisticated forms of interaction such as direct manipulation.

At that time, the idea of object-oriented programming arose and quickly gained much attention. New programming languages such as Simula and Smalltalk emerged and embraced this principle. After a short time, objects were also considered as distributable units for distributed middleware systems.

The following era of desktop computing was shaped by personal computers, growing microprocessor clock speeds, graphical user interfaces and desktop applications. At the same time, the old idea of non-linear information networks was rediscovered and its first real implementation appeared. Dating back to Vannevar Bush's MEMEX device and later Ted Nelson's concept of documents interconnected through hypertext, the World Wide Web represented a truly novel distributed information architecture in the Internet.

With the first commercial usage, the World Wide Web scored an instant success and soon hit a critical mass of users to become the most popular service within the whole Internet. This also motivated a complete transformation of our information economy. Technological advances at that time predicted the rise of mobile computing that should eventually proceed into an era of ubiquitous computing. Computers were not only going to become smaller and smaller, but also omnipresent and produced in various sizes and shapes. This introduced the notion of calm technology—technology that immerses into everyday life. Ubiquitous computing also heavily changes the way we are interacting with computers. Multi-modal and implicit interactions are favored and provide more natural interactions. Devices such as laptops, tablet computers, pad computers, mobile phones and smartphones have already blurred the boundaries between different types of computing devices. Combined with wireless broadband internet access, they have introduced new forms of mobility, providing connectivity at all times.

While we are currently on the verge of ubiquitous computing, there are other trends as well that are influencing the way we think about and do computing right now. The progress of microprocessors has saturated in terms of clock cycles due to physical constraints. Instead, modern CPUs are equipped with increasing numbers of cores. This trend has forced developers, architects and language designers to leverage multi-core architectures. The web has already started to oust desktop applications. Modern browsers are becoming the new operating systems, providing the basic runtime environment for web-based applications and unifying various platforms. The web is changing and provides continuously more user-centric services. Being able to handle and process huge numbers of users is common for social platforms such as Facebook and Twitter and corporations like Amazon or Google. Hosting such applications challenges traditional architectures and infrastructures. Labeled as so-called "Cloud Computing", highly available, commercial architectures emerged. They are built on large clusters of commodity hardware and enable applications to scale with varying demand over time on a pay-per-use model.

Now let us take a step back and summarize a few important developments:

1. The web is a dominant and ubiquitous computing technology. It will replace many traditional desktop application environments and provide ubiquitous information access.
2. Web applications have to cope with increasing demand and scale to larger user bases. Applications that incorporate features such as collaboration and web real-time interaction are facing new challenges as compared to traditional web applications.
3. Multi-core and multiprocessor architectures will dominate the processor landscape. At least for the next decades, performance gains of processors will mostly be attributed to increasing numbers of cores or processors and not to increased clock cycle frequencies.
4. Large, scalable systems can only be designed when taking into account the essence of distributed and concurrent systems. Appropriate programming languages, frameworks and libraries are necessary to implement such systems.

In this thesis, we will bring together these individual developments to have a comprehensive analysis of a particular challenge: How can we tackle concurrency when programming scalable web architectures?

1.2 Scope of this Thesis

Concurrency is an essential part of network services, and it is of outstanding importance for scalable web architectures. Thus we will have a detailed look on concurrency in three distinct areas of web architectures—connection handling, application logic and backend persistence. For each stage, we will discuss its main challenges and issues and explain existing approaches to tackle concurrency. We will then compare and assess the different solutions, while illustrating their advantages and disadvantages.

By dedicating our analysis to concurrent programming for scalable web architectures and their applications, to some extent we will also have a look at general concurrency techniques. However, topics such as parallel programming, parallel algorithms and high performance computing are out of scope for this review. We will also focus on pragmatic concurrency approaches instead of theoretical concurrency models. In a later chapter, we will dare to take a glance at possible concurrency concepts in future programming languages.

Furthermore, we will solely consider scalability in regard to concurrency, not taking into account other design and architectural decisions in detail.

1.3 Methodology of the Study

This thesis brings together different resources on concurrency, web architectures and scalable network infrastructures. The primary sources are research publications and technical analy-

ses. Additionally, documentations of design patterns, systems, programming languages and frameworks are considered.

As the main contribution of this thesis, we provide a comprehensive survey on the myriads of different concepts and techniques of concurrency inside web architectures. We compare and evaluate approaches based on their characteristic features and impacts, and provide guidelines for choosing the right approach for a given problem.

1.4 Road Map

We start off with a brief introduction to the World Wide Web, concurrency and scalability in chapter 2. We then focus on web architectures in chapter 3 and elaborate a general architectural model for scalable web architectures based on existing systems. Next, we have an isolated look at different stages of this model and get to know their relevant concurrency challenges and available approaches to adopt. This includes web servers in chapter 4, applications and business logic in chapter 5 and backend storages in chapter 6. In chapter 7, we provide a guideline for choosing a decent approach or concept for different scenarios. Next, we discuss the results of our considerations as part of chapter 8 and take a look at the essence of concurrency and scalability in distributed systems. We finally dare an outlook on the near future of web architectures and how it will be affected by new web technologies and programming concepts in chapter 9. The conclusion in chapter 10 sums up our survey on concurrency in web architectures.

2 The World Wide Web, Concurrency and Scalability

This chapter introduces the basic concepts of the World Wide Web, web applications, concurrency and scalability. It thus lays the foundation for our later analysis in the subsequent chapters 4,5 and 6.

2.1 The World Wide Web

The Internet and the World Wide Web (WWW) have become the backbone of our modern society and economy. The WWW is a distributed system of interlinked hypermedia resources in the Internet. It is based on a set of different, related technologies. We will have a brief look on the most important protocols and formats in this section, including Uniform Resource Identifiers (URIs), HTTP and HTML.

2.1.1 Uniform Resource Identifiers

URIs, currently specified in RFC 3986 [BL05], are strings that are used to reference resources. In terms of distributed systems, a URI has three distinct roles—naming, addressing, and identifying resources. We will focus on URIs identifying resources in the WWW, although they can be used for other abstract or physical entities as well. According to the specification, a URI consists of five parts: *scheme*, *authority*, *path*, *query* and *fragment*. However, only *scheme* and *path* are mandatory, the other parts are optional. *Scheme* is declaring the type of the URI and thus determines the meaning of the other parts of the URI. If used, *authority* points to the responsible authority of the referenced resource. In case of `http` as scheme, this part becomes mandatory and contains the host of the web server hosting the resource. It can optionally contain a port number (80 is implied for `http`) and authentication data (deprecated). The mandatory *path* section is used to address the resource within the scope of the *scheme* (and *authority*). It is often structured hierarchically. The optional *query* part provides non-hierarchical data as part of the resource identifier. *Fragment*s can be used to point to a certain part within the resource. The following example is adapted from RFC 3986 [BL05] and makes use of all five parts:

```
      http://example.com:8080/over/there?search=test#first
       \_/   \______________/\_________/ \_________/ \__/
        |           |            |            |        |
     scheme     authority       path        query   fragment
```

It identifies a web resource (scheme is `http`), that is hosted on the `example.com` (on port 8080). The resource path is `/over/there` and the query component contains the key/value pair `search=test`. Furthermore, the `first` fragment of the resource is referenced.

2.1.2 The Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is an application-level protocol that represents the foundation of communication for the WWW on top of TCP/IP. HTTP, as defined in RFC 2616 [Fie99], is a stateless protocol and complies with a client/server architecture and a request/response communication model. Servers host resources that are identified by URIs and can be accessed by clients. The client issues an HTTP request to the server which in return provides an HTTP response. The communication model limits the possible message patterns to single request/response cycles that are always initiated by the client. Apart from clients and servers, HTTP also describes optional intermediaries, so called proxies. These components provide additional features such as caching or filtering. Proxies combine features of a client and a server and are thus often transparent for the clients and servers in terms of communication.

HTTP requests and responses have a common structure. Both start with a request line respectively status line. The next part contains a set of header lines that include information about the request respectively response and about the entity.

The entity is an optional body of an HTTP message that contains payload such as a representation of the resource. While the first two parts of an HTTP message are text-based, the entity can be any set of bytes. HTTP request lines contain a request URI and a method. There are different HTTP methods that provide different semantics when applied to a resource, as shown in table 2.1.

In the subsequent HTTP response, the server informs the client about the outcome of a request by using predefined status codes. The classes of status codes can be seen in table 2.2. A simple request/response exchange shown in listing 2.1 as example. We will now have a closer look at two advanced features of HTTP that are interesting for our later considerations, namely connection handling and chunked encoding.

HTTP Connection Handling

As already mentioned, HTTP uses TCP/IP as underlying transport protocol. We will now examine the exact usage of TCP sockets for HTTP requests. The previous specifications of HTTP have suggested a separate socket connection for each request/response cycle. Adding the overhead of establishing a TCP connection for each request leads to poor performance and missing reusability of existing connections. The non-standard `Connection: Keep-Alive`

Method	Usage	Safe	Idempotent	Cachable
`GET`	This is the most common method of the WWW. It is used for fetching resource representations.	✓	✓	✓
`HEAD`	Essentially, this method is the same as `GET`, however the entity is omitted in the response.	✓	✓	✓
`PUT`	This method is used for creating or updating existing resources with new representations.		✓	
`DELETE`	Existing resources can be removed with `DELETE`		✓	
`POST`	`POST` is used to create new resources. Due to its lack of idempotence and safety, it also often used to trigger arbitrary actions.			
`OPTIONS`	Method provides meta data about a resource and available representations.	✓	✓	

Table 2.1: A table of the official HTTP 1.1 methods. In terms of RFC 2616 a method is *safe*, when a request using this method does not change any state on the server. If multiple dispatches of a request result in the same side effects than a single dispatch, the request semantics is called *idempotent*. If a request method provides *cacheability*, clients may store responses according to the HTTP caching semantics.

header was a temporary workaround, but the current HTTP 1.1 specification has addressed this issue in detail. HTTP 1.1 introduced *persistent connections* as default. That is, the underlying TCP connection of an HTTP request is reused for subsequent HTTP requests. *Request pipelining* further improves throughput of persistent connections by allowing to dispatch multiple requests, without awaiting for responses to prior requests. The server then responds to all incoming request in the same sequential order. Both mechanisms have improved the performance and decreased latency problems of web applications. But the management of multiple open connections and the processing of pipelined requests has revealed new challenges for web servers as we will see in chapter 4.

HTTP Chunked Transfer Encoding

An HTTP message must contain the length of its entity, if any. In HTTP 1.1, this is neccessary for determining the overall length of a message and detecting the next message of a persistent connection. Sometimes, the exact length of an entity cannot be determined a priori. This is

Range	Status Type	Usage	Example Code
1xx	informational	Preliminary response codes	100 Continue
2xx	success	The request has been successfully processed.	200 OK
3xx	redirection	The client must dispatch additional requests to complete the request.	303 See Other
4xx	client error	Result of an erroneous request caused by the client.	404 Not Found
5xx	server error	A server-side error occured (not caused by this request being invalid).	503 Service Unavailable

Table 2.2: A table of the code ranges of HTTP response codes. The first digit determines the status type, the last two digits the exact response code.

```
GET /html/rfc1945 HTTP/1.1
Host: tools.ietf.org
User-Agent: Mozilla/5.0 (Ubuntu; X11; Linux x86_64; rv:9.0.1) Gecko/20100101
     Firefox/9.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de-de,de;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
If-Modified-Since: Sun, 13 Nov 2011 21:13:51 GMT
If-None-Match: "182a7f0-2aac0-4b1a43b17a1c0;4b6bc4bba3192"
Cache-Control: max-age=0

HTTP/1.1 304 Not Modified
Date: Tue, 17 Jan 2012 17:02:44 GMT
Server: Apache/2.2.21 (Debian)
Connection: Keep-Alive
Keep-Alive: timeout=5, max=99
Etag: "182a7f0-2aac0-4b1a43b17a1c0;4b6bc4bba3192"
Content-Location: rfc1945.html
Vary: negotiate,Accept-Encoding
```

Listing 2.1: Example HTTP request/response

especially important for content that is generated dynamically or for entities that are compressed on-the-fly. Therefore, HTTP provides alternative transfer encodings. When *chunked transfer encoding* is used, the client or server streams chunks of the entity sequentially. The length of the next chunk to expect is prepended to the actual chunk. A chunk length of 0 denotes the end of the entity. This mechanism allows the transfer of generated entities with arbitrary length.

2.1.3 Web Formats

HTTP does not restrict the document formats to be used for entities. However, the core idea of the WWW is based on hypermedia, thus most of the formats are hypermedia-enabled. The single most important format is the Hypertext Markup Language (HTML) and its descendants.

Hypertext Markup Language

HTML [Jac99] is a markup language derived from the Standard Generalized Markup Language (SGML) [fS86] and influenced by the Extensible Markup Language (XML) [Bra08]. HTML provides a set of elements, properties and rules for describing web pages textually. A browser parses the HTML document, using its structural semantics for rendering a visual representation for humans. HTML supports hypermedia through hyperlinks and interactive forms. Also, media objects such as images can be used in an HTML document. The appearance and style of an HTML document can be customized by using Cascading Style Sheets (CSS) [Mey01]. For more dynamic user interfaces and interactive behavior, HTML documents can be enriched with embedded code of scripting languages, such as *JavaScript* [ECM99]. For instance, it can be used to programmatically load new contents in the background, without a complete reload of the page. This technique, also known as Asynchronous JavaScript and XML (AJAX), has been one of the keys for more responsive user interfaces. It thus enables web applications to resemble interfaces of traditional desktop applications.

HTML5

The fifth revision of the HTML standard [Hya09] introduces several markup improvements (e.g. semantic tags), better multimedia content support, but most notably a rich set of new APIs. These APIs address various features including client-side storage, offline support, device sensors for context awareness and improved client-side performance. The Web Sockets API [Hic09a] complements the traditional HTTP request/response communication pattern with a low latency, bidirectional, full-duplex socket based on the WebSocket protocol [Fet11]. This is especially interesting for real-time web applications.

Generic Formats

Besides proprietary and customized formats, web services often make use of generic, structured formats such as XML and JavaScript Object Notation (JSON) [Cro06]. XML is a comprehensive markup language providing a rich family of related technologies, like validation, transformation or querying. JSON is one of the leightweight alternatives that focuses solely on the succinct representation of structured data. While there is an increasing interest in leightweight formats for web services and messages, XML provides still the most extensive tool set and support.

2.2 Web Applications

There is a huge number of different applications available in the web. In the following, we distinguish two separate types of web applications—*web sites* and *web services*. Web sites are web-based applications that are designed for humans and browser-based access. By contrast, web services are web-based interfaces for machine-to-machine communication. Although this distinction is rather arbitrary, it will help us to identify different requirements and features.

2.2.1 Web Sites

Web sites have evolved from hyper-referenced, text-based research documents to highly interactive, social and collaborative applications for many different purposes. Web content has become increasingly dynamic and based on content provided by the users. Web technologies such as JavaScript (JS), AJAX, and HTML5 have introduced more interactive user interfaces and boosted this transition. Thanks to powerful APIs provided by modern browsers, web applications are already starting to replace traditional desktop applications and native mobile applications.

Examples

We will now introduce some popular types for web sites that are interesting in terms of scalability and concurrency.

Social Network Sites Social networks are sites that transfer social interactions to the web. They often try to reflect real world social relations of its users (e.g. Facebook[1]) or focus on specific topics (e.g. dopplr for travelling[2]). Social network sites motivate their users to interact, for instance via instant messaging. Also, social networks heavily rely on user generated content and context-specific data, such as geo-tagged content. User content and actions are often published into activity streams, providing a "real-time" feed of updates that can be watched live by other users.

1 http://facebook.com/
2 http://www.dopplr.com/

Collaborative Web Applications These web applications allow a group of people to collaborate via the web. Traditional examples are wiki systems, where users can collectively edit versions of text documents. More recent collaborative applications incorporate soft real-time aspects. For example, Etherpad[1] is a web-based word processor that supports multiple users working on the same document concurrently.

E-Commerce Sites E-Commerce sites such as Amazon[2] are traditional commercial sites in the web selling products online. Interestingly enough, many sites have adopted features known from social sites for business purposes. By commenting, rating and tagging products, users can participate on these sites beyond just buying items. User-generated content is then used to cluster product items and compute accurate recommendations. Thus, commercial sites face similar challenges to social sites to some extent.

2.2.2 Web Services

Web services provide access to application services using HTTP. Thus, web services often resemble traditional mechanisms for distributed computing such as Remote Procedure Call (RPC) or message passing, though based on web technologies.

Opposed to web sites, web services are not targeting direct (human) user access in a first place. Instead, web services enable machine-to-machine communication and provide application features via an interface and structured messages. Several web applications provide both, a web site and a web service interface (API). While the web site is used for browser-based access, the web service can be used for custom applications such as mobile client applications or scripted program-based service interactions.

XML-RPC

XML-RPC has been one of the first attempts to transfer traditional RPC-based services to the web. It makes use of HTTP POST requests for dispatching procedure calls and an XML-based serialization format for call parameters and return values. Listing 2.2 provides an example taken from the original specification [Win99].

It is important to clarify that XML-RPC is using HTTP as a generic transport protocol for RPC calls. It does not take advantage of any HTTP features such as caching, status codes for error handling or header fields for negotiation.

1 http://etherpad.org/
2 http://www.amazon.com

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-Length: 181

<?xml version="1.0"?> <methodCall> <methodName>examples.getStateName</
    methodName> <params> <param> <value><i4>41</i4></value> </param> </
    params> </methodCall>

HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT

<?xml version="1.0"?> <methodResponse> <params> <param> <value><string>South
     Dakota</string></value> </param> </params> </methodResponse>
```

Listing 2.2: Example XML-RPC call

SOAP, WSDL and WS-*

The stack of SOAP [Gud07], WSDL [Liu07], UDDI [Cle04] and a myriad of additional extensions (WS-*) forms a more comprehensive approach for machine-to-machine communication, mostly based on web technologies. It is widely used and particularly popular in enterprise environments. As previously mentioned, SOAP is a successor of XML-RPC. It specifies the format, call semantics and exchange patterns of XML-encoded messages between parties. Various extension specifications, often labeled as `WS-*`, address additional features of SOAP-based web services such as security or orchestration. The Web Services Description Language (WSDL) is another important specification for this kind of web services. It provides XML-based, machine-readable service descriptions, comparable to interface definition languages of traditional RPC protocols. Universal Description, Discovery and Integration (UDDI) was originally a third component providing registry functions for web services, but it has almost entirely lost its significance in practice.

Although the SOAP/WSDL stack is generally known as *web* service stack, it dismisses parts of the original idea of the web. For instance, the stack uses HTTP as a pure communication protocol for exchanging messages, that can be replaced by other protocols. Similarly to XML-RPC, this web service stack does not use HTTP as an application-level protocol.

Representational State Transfer

Representational State Transfer (REST) is an architectural style that embraces the fundamentals of the WWW. It has been introduced as part of the doctoral thesis of Roy T. Fielding [Fie00],

who was also responsible for the HTTP specifications to some extent. Not surprisingly, the REST style takes full advantage of HTTP features. Fielding suggests a resource-oriented architecture that is defined by a set of constraints:

Client-server The client-server style is motivated by a separation of concerns. It promotes to separate user interfaces and data storage of the system. It also simplifies dedicated component design and allows an independent evolution of client and server components.

Statelessness Statelessness promotes scalability and reliability. It also makes the server-side development easier.

Cache Optional non-shared caching is a constraint that helps to reduce latency and improves scalability and efficiency.

Layered System Using a layered system enables a set of interesting architectural extensions such as legacy encapsulation, load balancing and (shared) caching.

Code-on-Demand This optional constraint provides extensibility by simplifying client updates.

Uniform Interface The uniform interface is the most important constraint of the REST style and determines the resources as central entities that can be operated on using a fixed set of verbs (the HTTP methods). This constraint can be divided into four sub-constraints:

- *Identification of resources* is accomplished by the usage of URIs inside a RESTful web service.
- *Manipulation via representations* promotes the change of resources through the submission of altered representations by the client.
- *Self-descriptive messages* provide enough information so that they can be processed without further external knowledge.
- Most importantly, *hypermedia as the engine of application state* promotes the usage of hypermedia inside the representations. State transitions are triggered by accessing resources through their uniform interface and by following hyperlinks inside their representations.

The broad interest in REST and its recent popularity is not the only reason it is listed here. The conceptual ideas behind REST are taken from HTTP and based on the lessons learned from the WWW. The ideas promote loose coupling, stateless communication, reusability and interoperability. Thus, they establish a great basis for scalability of architectures that are complying with the REST style.

2.3 Concurrency

Concurrency is a property of a system representing the fact that multiple activities are executed at the same time. According to Van Roy [Roy04], a program having "several independent activities, each of which executes at its own pace". In addition, the activities may perform some kind of interaction among them. The concurrent execution of activities can take place in different environments, such as single-core processors, multi-core processors, multi-processors or even on multiple machines as part of a distributed system. Yet, they all share the same underlying challenges: providing mechanisms to control the different flows of execution via coordination and synchronization, while ensuring consistency.

Apart from recent hardware trends towards multi-core and multiprocessor systems, the use of concurrency in applications is generally motivated by performance gains. Cantrill et al. [Can08] describe three fundamental ways how the concurrent execution of an application can improve its performance:

Reduce latency A unit of work is executed in shorter time by subdivision into parts that can be executed concurrently.

Hide latency Multiple long-running tasks are executed together by the underlying system. This is particularly effective when the tasks are blocked because of external resources they must wait upon, such as disk or network I/O operations.

Increase throughput By executing multiple tasks concurrently, the general system throughput can be increased. It is important to notice that this also speeds up independent sequential tasks that have not been specifically designed for concurrency yet.

The presence of concurrency is an intrinsic property for any kind of distributed system. Processes running on different machines form a common system that executes code on multiple machines at the same time.

Conceptually, all web applications can be used by various users at the same time. Thus, a web application is also inherently concurrent. This is not limited to the web server that must handle multiple client connections in parallel. Also the application that executes the associated business logic of a request and the backend data storage are confronted with concurrency.

2.3.1 Concurrency and Parallelism

In the literature, there are varying definitions of the terms concurrency and parallelism, and their relation. Sometimes both terms are even used synonymously. We will now introduce a distinction of both terms and of their implications on programming.

Concurrency vs. Parallelism

Concurrency is a conceptual property of a program, while parallelism is a runtime state. Concurrency of a program depends on the programming language and the way it is coded, while parallelism depends on the actual runtime environment. Given two tasks to be executed concurrently, there are several possible execution orders. They may be performed sequentially (in any order), alternately, or even simultaneously. Only executing two different tasks simultaneously yields true parallelism. In terms of scheduling, parallelism can only be achieved if the hardware architecture supports parallel execution, like multi-core or multi-processor systems do. A single-core machine will also be able to execute multiple threads concurrently, however it can never provide true parallelism.

Concurrent Programming vs. Parallel Programming

Differentiating concurrent and parallel programming is more tedious, as both are targeting different goals on different conceptual levels. Concurrent programming tackles concurrent and interleaving tasks and the resulting complexity due to a nondeterministic control flow. Reducing and hiding latency is equally important to improving throughput. Instead, parallel programming favors a deterministic control flow and mainly reaches for optimized throughput. The internals of a web server are the typical outcome of concurrent programming, while the parallel abstractions such as Google's `MapReduce` [Dea08] or Java's `fork/join` [Lea00] provide a good example of what parallel programming is about. Parallel programming is also essential for several specialized tasks. For instance, a graphics processing unit is designed for massive floating-point computational power and usually runs a certain numerical computation in parallel on all of its units at the same time. High-performance computing is another important area of parallel programming. It takes advantage of computer clusters and distributes sub-tasks to cluster nodes, thus speeding up complex computations.

Computer Architectures Supporting Concurrency

The previous differentiation can also be backed by a closer look on the architectures where physical concurrency is actually possible. We will therefore refer to Flynn's taxonomy [Fly72], which is shown in table 2.3. This classification derives four different types of computer architectures, based on the instruction concurrency and the available data streams. The Single Instruction,

	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Table 2.3: Flynn's taxonomy classifying different computer architectures.

Single Data stream (SISD) class is represented by traditional single processor machines. We do not consider them further due to their lack of physical concurrency. Multiple Instruction, Single Data stream (MISD) is a rather exotic architecture class, sometimes used for fault-tolerant computing where the same instructions are executed redundantly. It is also not relevant for our considerations. Real parallelism can only be exploited on architectures that support multiple data streams—Single Instruction, Multiple Data streams (SIMD) and Multiple Instruction, Multiple Data streams (MIMD). SIMD represents the aforementioned architecture class targeting dedicated parallel executions of computations such as graphics processing unit and vector processors. SIMD exploits data-level parallelism which is not suitable for handling web requests. Accordingly, this type of architecture is not relevant for our consideration. We will focus on the last remaining class, MIMD. It represents architectures that rely on a shared or distributed memory and thus includes architectures possessing multiple cores, multiple CPUs or even multiple machines. In the following, we will primarily focus on concurrent programming (parallel execution of subtasks is partially relevent in chapter 5) and only on the MIMD class of architectures.

2.3.2 Models for Programming Concurrency

Van Roy [Roy04] introduces four main approaches for programming concurrency that we will examine briefly (see figure 2.1). The more important models will be studied later as potential solutions for programming concurrency inside web architectures, especially in chapter 5.

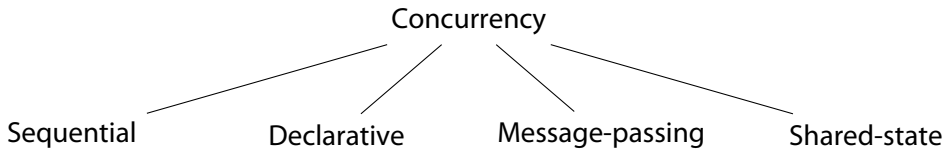

Figure 2.1: Models for Programming Concurrency (Van Roy [Roy04])

Sequential Programming

In this deterministic programming model, no concurrency is used at all. In its strongest form, there is a total order of all operations of the program. Weaker forms still keep the deterministic behaviour. However, they either make no guarantees on the exact execution order to the programmer *a priori*. Or they provide mechanisms for explicit preemption of the task currently active, as co-routines do, for instance.

Declarative Concurrency

Declarative programming is a programming model that favors implicit control flow of computations. Control flow is not described directly, it is rather a result of computational logic of the

program. The declarative concurrency model extends the declarative programming model by allowing multiple flows of executions. It adds implicit concurrency that is based on a data-driven or a demand-driven approach. While this introduces some form of nondeterminism at runtime, the nondeterminism is generally not observable from the outside.

Message-passing Concurrency

This model is a programming style that allows concurrent activities to communicate via messages. Generally, this is the only allowed form of interaction between activities which are otherwise completely isolated. Message passing can be either synchronous or asynchronous resulting in different mechanisms and patterns for synchronization and coordination.

Shared-state Concurrency

Shared-state concurrency is an extended programming model where multiple activities are allowed to access contended resources and states. Sharing the exact same resources and states among different activities requires dedicated mechanisms for synchronization of access and coordination between activities. The general nondeterminism and missing invariants of this model would otherwise directly cause problems regarding consistency and state validity.

2.3.3 Synchronization and Coordination as Concurrency Control

Regardless of the actual programming model, there must be an implicit or explicit control over concurrency (at least within the runtime environment). It is both hazardous and unsafe when multiple flows of executions simultaneously operate in the same address space without any kind of agreement on ordered access. Two or more activities might access the same data and thus induce data corruption as well as inconsistent or invalid application state. Furthermore, multiple activities that work jointly on a problem need an agreement on their common progress. Both issues represent fundamental challenges of concurrency and concurrent programming.

Synchronization and *coordination* are two mechanisms attempting to tackle this. Synchronization, or more precisely *competition synchronization* as labeled by Sebesta [Seb05], is a mechanism that controls access on shared resources between multiple activities. This is especially important when multiple activities require access to resources that cannot be accessed simultaneously. A proper synchronization mechanism enforces exclusiveness and ordered access on the resource by different activities. Coordination, sometimes also named *cooperation synchronization* (Sebesta [Seb05]), aims at the orchestration of collaborating activities.

Synchronization and coordination are sometimes conflated in practice. Both mechanisms can be either implicit or explicit (Van Roy [Roy04]). *Implicit synchronization* hides the synchronization as part of the operational semantics of the programming language. Thus, it is not part of the visible program code. On the contrary, *explicit synchronization* requires the programmer to add explicit synchronization operations to the code.

2.3.4 Tasks, Processes and Threads

So far, our considerations of concurrency were based on computer architectures and programming models. We will now show how they interrelate by introducing the actual activity entities used and provided by operating systems and how they are mapped to the hardware. Note that we will use the term *task* as a general abstraction for a unit of execution from now on.

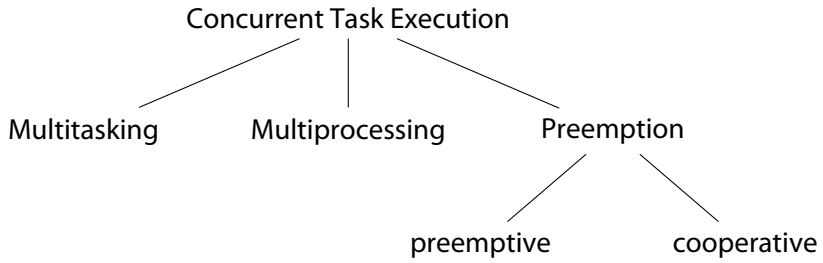

Figure 2.2: Orthogonal concepts for the concurrent execution of multiple tasks.

The ability to execute multiple tasks concurrently has been a crucial requirement for operating systems. It is addressed by *multitasking*. This mechanism manages an interleaved and alternating execution of tasks. In case of multiple CPU cores or CPUs, multitasking can be complemented by *multiprocessing*, which allocates different tasks on available cores/CPUs. The key concept for both mechanisms is *scheduling*, which organizes the assignment of processing time to tasks using *scheduling strategies*. Appropriate strategies can have different aims, such as fair scheduling between tasks, fixed latencies for task executions or maximum utilization. Another distinction of scheduling strategies is their preemption model. In the *preemptive* model, the scheduler assigns processing time to a task and eventually revokes it. The task has no control over the preemption. In a *cooperative* model, the task itself has the responsibility of yielding after some time to allow another task to run. Scheduling is an important duty of any operating system. However, it is also noteworthy that applications themselves can provide some kind of scheduling of their own internal tasks, as we will see in chapter 4 and 5.

Operating systems generally provide different types of tasks—processes and threads. Essentially, they represent different task granularities. A *process* is a heavyweight task unit and owns system resources such as memory and file descriptors that are allocated from the operating system. *Threads* are leightweight task units that belong to a certain process. All threads allocated within the same process share memory, file descriptors and other related resources. Creating threads is a relatively cheap operation compared to the creation of new processes.

Most concurrent applications make heavy use of multiple threads. However, this does not imply the availability of threads as explicit entities of the programming language itself. Instead, the execution environment might map other concurrency constructs of a programming language to actual threads at runtime. Similarly, the shared-state property of multithreading might be idiomatically hidden by the language.

2.3.5 Concurrency, Programming Languages and Distributed Systems

Next, we consider the strong relationship between concurrent programming, programming languages and distributed systems when building large architectures. Programming distributed systems introduces a set of additional challenges compared to regular programming. The "Fallacies of Distributed Computing" [RGO06] provide a good overview on some of the important pitfalls that must be addressed. For instance, not anticipating errors in a network or ignoring the different semantics of local and remote operation are common misconceptions that are described.

From a software engineering perspective, the major challenges are fault tolerance, integration of distribution aspects and reliability. As we have already seen before, distributed systems are inherently concurrent and parallel, thus concurrency control is also essential.

Programming languages to be used for distributed systems must either incorporate appropriate language idioms and features to meet these requirements. Otherwise, frameworks are necessary to provide additional features on top of the core language.

Ghosh et al. [Gho11] have considered the impact of programming languages on distributed systems. They pointed out that mainstream languages like Java and C++ are still the most popular choice of developing distributed systems. They are combined with middleware frameworks most of the time, providing additional features. However, the strengths of general purpose languages do not cover the main requirements of distributed systems to a great extent. The experiences with RPC-based systems (see Kendall et al. [Ken94]) and their object-based descendents (see Vinoski [Vin08]) have raised some questions to this approach. Middleware systems providing distributability compensate for features missing at the core of a language. Thus, the systems actually meet the necessary requirements, but they are often also cumbersome to use and introduce superfluous complexity.

Recently, there has been an increasing interest in various alternative programming languages embracing high-level concurrency and distributed computing. Being less general, these languages focus on important concepts and idioms for distributed systems, such as component abstractions, fault tolerance and distribution mechanisms. It is interesting for our considerations that most of these languages oriented towards distributed computing also incorporate alternative concurrency approaches. We will have a brief look on some of these languages as part of chapter 5.

2.4 Scalability

Scalability is a non-functional property of a system that describes the ability to appropriately handle increasing (and decreasing) workloads. According to Coulouris et al. [Dol05], "a system is described as scalable, if it will remain effective when there is a significant increase in the number of resources and the number of users". Sometimes, scalability is a requirement that necessitates the usage of a distributed system in the first place. Also, scalability is not to be confused with raw speed or performance. Scalability competes with and complements other non-functional requirements such as availability, reliability and performance.

2.4.1 Horizontal and Vertical Scalability

There are two basic strategies for scaling—vertical and horizontal. In case of vertical scaling, additional resources are added to a single node. As a result, the node can then handle more work and provides additional capacities. Additional resources include more or faster CPUs, more memory or in case of virtualized instances, more physical shares of the underlying machine. In contrast, horizontal scaling adds more nodes to the overall system.

Both scaling variants have different impacts on the system. Vertical scaling almost directly speeds up the system and rarely needs special application customizations. However, vertical scaling is obviously limited by factors such as cost effectiveness, physical constraints and availability of specialized hardware. Horizontal scaling again requires some kind of inherent distribution within the system. If the system cannot be extended to multiple machines, it could not benefit from this type of scaling. But if the system does support horizontal scaling, it can be theoretically enlarged to thousands of machines. This is the reason why horizontal scaling is important for large-scale architectures. Here, it is common practice to focus on horizontal scaling by deploying lots of commodity systems. Also, relying on low cost machines and anticipating failure is often more economical than high expenses for specialized hardware.

Considering a web server, we can apply both scaling mechanisms. The allocation of more available system resources to the web server process improves its capacities. Also, new hardware can provide speed ups under heavy load. Following the horizontal approach, we setup additional web servers and distribute incoming requests to one of the servers.

2.4.2 Scalability and other Non-functional Requirements

In software engineering, there are several important non-functional requirements for large software architectures. We will consider operational (runtime) requirements related to scalability: high availability, reliability and performance. A system is available, when it is capable of providing its intended service. *High availability* is a requirement that aims at the indentured availability of a system during a certain period. It is often denoted as percentiles of uptime, restricting the maximum time to be unavailable.

Reliability is a closely related requirement that describes the time span of operational behavior, often measured as meantime between failures. Scalability, anticipating increasing load of a system, challenges both requirements. A potential overload of the systems due to limited scalability harms availability and reliability. The essential technique for ensuring availability and reliability is redundancy and the overprovisioning of resources. From a methodical viewpoint, this is very similar to horizontal scaling. However, it is important not to conflate scalability and availability. Spare resources allocated for availability and failover can not be used for achieving scalability at the same time. Otherwise, only one requirement can be guaranteed at once.

Performance of software architectures can have multiple dimensions such as short response times (low latencies) and high throughput along with low utilization. Again, increasing load

of an application may affect the requirement negatively. Unless an application is designed with scalability in mind and there are valid scaling options available, performance may degrade significantly under load.

Note that in most web architecture scenarios, all of the requirements mentioned are desirable. However, especially when resources are limited, there must be some kind of trade-off favoring some of the requirements, neglecting others.

2.4.3 Scalability and Concurrency

The relation between scalability and concurrency is twofold. From one perspective, concurrency is a feature that can make an application scalable. Increasing load is opposed to increasing concurrency and parallelism inside the application. Thanks to concurrency, the application stays operational and utilizes the underlying hardware to its full extent. That is, above all, scaling the execution of the application among multiple available CPUs/cores.

Although it is important to differentiate between increased performance and scalability, we can apply some rules to point out the positive impacts of parallelism for scalability. Certain problems can be solved faster when more resources are available. By speeding up tasks, we are able to conduct more work at the same time. This is especially effective when the work is composed of small, independent tasks.

We will now have a look at a basic law that describes the speed-up of parallel executions. Amdahl's law [Goe06], as seen in equation 2.1, describes the maximum improvement of a system to expect when resources are added to a system under the assumption of parallel execution. A key point hereof is the ratio of serial and parallel subtasks. N is the number of processors (or cores) available, and F denotes the fraction of calculations to be executed serially.

$$speedup \leq \frac{1}{F + \frac{1-F}{N}} \tag{2.1}$$

Note that in case of a web server, parallelizable tasks are predominant. However, highly interactive and collaborative web applications require increasing coordination between requests, weakening the isolated parallelism of requests.

From a different angle, concurrency mechanisms themselves have some kind of scalability property. That is basically the ability to support increasing numbers of concurrent activities or flows of execution inside the concurrency model. In practice, this involves the language idioms representing flows of executions and corresponding mappings to underlying concepts such as threads. For our considerations, it is also interesting to relate incoming HTTP requests to such activities. Or more precisely, how we allocate requests to language primitives for concurrency and what this implies for the scalability of the web application under heavy load.

2.4.4 Scalability of Web Applications and Architectures

It is the main scalability challenge of web applications and architectures to gracefully handle growth. This includes growth of request numbers, traffic and data stored. In general, increasing load is a deliberate objective that testifies increasing usage of the application. From an architectural point of view, we thus need so-called *load scalability*. That is the ability to adapt its resources to varying loads. A scalable web architecture should also be designed in a way that allows easy modification and upgrade/downgrade of components.

2.5 Summary

We have seen that the World Wide Web is a distributed system of interlinked hypermedia resources, based on URIs, HTTP and hypertext document formats like HTML. The persisting success of the web has pushed web-based applications into the limelight. Browsers are becoming the most used local applications for computers, providing access to a myriad of different applications via the web. Even in the mobile area, web applications designed for mobile usage successfully challenge native mobile apps.

Web applications designed for millions of users are faced with extraordinary challenges, including the reasonable handling of inherent parallelism and the profound ability to scale.

Hence, concurrency is not just a trait of a web architecture. It is also a mandatory and crucial principle when programming and implementing large-scale web applications in order to utilize hardware capacities to the fullest. As concurrent programming is a non-trivial procedure, we are in need of appropriate abstractions and usable programming models.

The scalability requirement of large web architectures obliges us to think about growth from the beginning. The provision of additional resources later on should provide us with the maximal increase in performance and power possbile for our application. As the vertical scalability of a single node is limited, we must take into account horizontal growth, which also brings in distribution compulsively.

3 The Quest for Scalable Web Architectures

This chapter features a look behind the architectures that provide web applications. Before we examine the more recent cloud-based architectures, we will outline several technologies that have emerged to serve dynamic web content. We will then see how these technologies have been integrated into common architectures. Afterwards, we go to load-balancing, an established and important mechanism for scalability in web architectures. We will then learn about the infrastructure and services provided by popular cloud vendors.

Based on that, we will come up with our own generic architectural model for scalable web applications as an important contribution of this chapter. The model is the foundation for our further considerations, focusing on concurrency. A separation into the different architectural components does not just provide a clearer scalability path for each component. It also helps us to reason about particular concurrency challenges inherent to the distinct components.

3.1 Traditional Web Architectures

Although current web architectures use almost the same application protocol as first web servers did, their internals have changed considerably. Especially the rise of dynamic web content has had a reasonable impact on architectural concepts. As the web has been growing, tiered architectures appeared that separate different responsibilities of architectural components. Growing architectures also demanded for ways of scaling web applications, and load-balancing has established itself as a decent mechanism. We now have a look at the integration of dynamic content into web applications and consequences for servers by giving an overview of different technologies. Then we examine the concept of tiered architectures and load-balancing.

3.1.1 Server-Side Technologies for Dynamic Web Content

In the early 90s, the first web servers were network servers that provided access solely to static files via HTTP. Within a short time, there was an increasing demand in more dynamic contents. For instance, enriching static HTML files with mutable server state, generating complete HTML

files on-the-fly or dynamically responding to form submissions was requested to improve the user experience. One way to do this was altering the web servers and including mechanisms for dynamic content creation deep inside the code of web servers. Of course, this was a cumbersome approach and conflated web server internals and web application programming. As a result, more general solutions were needed and soon emerged, as for instance CGI.

Common Gateway Interface

The Common Gateway Interface (CGI) [Rob04] is a standardized interface for delegating web requests to external applications that handle the request and generate a response. CGI can be used when the interface is supported by both the web server and the external application. In practice, most of these applications are implemented using scripting languages. For each incoming request against a URI mapped to a CGI application, the web server spawns a new process. This process executes the CGI application and provides specific variables such as request headers and server variables via environment variables. Request entities can be read by the CGI process via `STDIN`, and the the generated response (both headers and the response entity) are written to `STDOUT`. After generating the response, the CGI process terminates. Only using external processes and communication via environment variables, `STDIN` and `STDOUT` provides a simple interface. It allows any application to handle and generate web content when supporting these basic mechanisms. Especially Perl and other scripting languages such as PHP: Hypertext Preprocessor (PHP) have been used later extensively to build web applications based on CGI. These languages can be directly embedded into existing HTML markup, and the code added is executed on each request. Alternatively, they provide ways to generate HTML documents, often using template engines.

However, the CGI model has several problems, in particular scalability and performance. As we have seen previously, processes are heavyweight structures for tasks (see also 2.3.4). They require reasonable overhead and resources for creation. Thus, mapping each dynamic request to a new process to be spawned is a very costly operation. It not just increases the latency due to process creation, it also wastes server resources due to the spawning overhead and limits the number of concurrent request that can be handled. Given the fact that most CGI applications are script files that have to be interpreted on each execution, average latencies deteriorate even more. The communication via `STDIN`/`STDOUT` offers another severe problem. This way of communication between processes limits the distributability of components, since both processes must be located on the same machine. There is no way of decoupling both components in a distributed way when using CGI.

FastCGI

FastCGI mitigates the main issues of CGI by specifying an interface protocol to be used via local sockets or TCP connections. Thus, web servers and applications generating the responses are

decoupled and can be located on different machines. Making no restrictions on the concurrency model, the backend application can be implemented as a long-running process with internal multithreading. In effect, the overhead of per-request process creation is gone and the concurrency can be increased by a great extent.

Web Server Extension Modules

Another alternative to CGI are server extension modules. Instead of external interfaces, internal module interfaces of the web server are provided that allow to plug in modules. These modules are regularly used to embed interpreters for scripting languages into the server context. In particular, the concurrency model of the server is often applied to the script execution. As a result, request handling and dynamic content generation can be executed within the same thread or process. The tight integration into the server can improve performance and generally provides better speed results than CGI-based scripts. However, this model again prevents loose coupling and makes the separation of web server and backend application more difficult.

Web Application Containers

The original CGI model was not appropriate for some languages such as Java. The dedicated process-per-request model and the startup times of the JVM made it completely unusable. As a result, alternative approaches emerged, such as the Java Servlet specification. This standard specifies a container, that hosts and executes web applications and dispatches incoming requests to a pool of threads and corresponding objects for request handling. Special classes (`javax.servlet.Servlet`) are used that provide protocol-specific methods. The `HttpServlet` class provides methods such as `doGet` or `doPost` to encapsulate HTTP methods. JavaServer Pages (JSP) provide an alternative syntax and allows to inline code into HTML files. On startup, these files are then converted into regular Servlet classes automatically. The internal multithreading provides better performance and scalability results than CGI, and web application containers and web servers can also be decoupled. In this case, a connecting protocol like Apache JServ Protocol [Sha00] is needed.

3.1.2 Tiered Architectures

Patterns for remotely accessible, interactive applications and a separation of concerns such as the model-view-controller or the presentation–abstraction–control pattern have been developed a long time before the web has emerged. An important architectural pattern for web in this regard is the concept of a multi-tier architecture [Fow02]. It describes the separation of different components or component groups as part of a client-server architecture. This separation is often twofold—it either describes a logical decomposition of an application and its functionality. Or it describes a rather technical split of deployment components. There are also different granularities of this separation. Nominating tiers for dedicated purposes (e.g. business process management)

or further breaking down tiers (e.g. splitting data access and data storage) yields additional tiers. We will now have a look at the most common separation of web architectures, a logical separation of concerns into three distinct tiers.

Presentation tier The presentation tier is responsible for displaying information. In terms of a web application, this can be done either by a graphical user interface based on HTML (web sites), or by providing structured data representations (web services).

Application logic tier This tier handles the application logic by processing queries originated from the presentation tier and providing appropriate (data) responses. Therefore, the persistence tier is accessed. The application tier encapsulates business logic and data-centric functionalities of the application.

Persistence tier The last tier is used for storing and retrieving application data. The storage is usually persistent and durable, i.e. a database.

When mapping these logical tiers to application components, there are often a number of different possibilities. Traditional web applications allocate all tiers to the server side, expect for the rendering of HTML pages that takes place in the browser. This resembles a traditional thin client architecture. Modern browser technologies such as the Web Storage API or IndexedDB now allow applications to be located entirely within the client side, at least during offline usage. This temporarily pushes all conceptual tiers into the browser and resembles a fat client. For the most part of current web applications, tiers are balanced and presentation is mainly a task of the browser. Modern web applications often try to provide as much functionalities as possible on client side for better user experience, and rely on server-side functions in case of missing browser support. This is known as *graceful degradation*[1], a term borrowed from fault-tolerant system design [Ran78]. To some extent, application logic is also available on client side, but most functionality is on the server. Sometimes, features are also provided redundantly. This is especially important for security-critical tasks such as input validation. Persistence is assigned to the server side with a few exceptions such as temporarily offline usage.

Focusing on the server-side architecture, the tiers provide a basic set of components. Components for the presentation, application and persistence tier can be placed on a single machine, or deployed to dedicated nodes. We will elaborate a more detailed architectural model based on components in section 3.3.

3.1.3 Load-Balancing

The limitations of vertical scaling force us to deploy multiple web servers at a certain scale. We thus need a mechanism of balancing workload from incoming requests to multiple available servers.

1 http://accessites.org/site/2007/02/graceful-degradation-progressive-enhancement/

As a result, we want an effective resource utilization of all servers (this is the primary target of load-balancing, and not high availability as we have seen in subsection 2.4.2). Handling a request is a rather short-living task, but the huge number of parallel requests makes the appropriate allocation and distribution of requests to servers a decent challenge. Several strategies have been developed to address this, as we will see soon. When implementing a load-balancing solution, another decision concerns the technical implementation level of connection forwarding [Sch06].

Network-level vs. Application-level Balancing

In HTTP, web servers are distinguished by hostname and port. Hostnames are resolved to IP addresses using the DNS protocol. However, a single IP address cannot be assigned to multiple online hosts at the same time. A first way of mapping a single hostname to multiple servers is a DNS entry that contains multiple IP addresses and keeps a rotating list. In practice, this is a naive load-balancing approach, as DNS has several unwanted characteristics such as the difficult removal of a crashed server or the long dissemination times for updates. Frequently changing hostname-to-IP resolutions interferes with secured connections via SSL/TLS. While DNS-based balancing can help to some extent (e.g. balancing between multiple load-balancers), we generally need more robust and sophisticated mechanisms. With reference to the ISO/OSI model, both the application layer and lower-level layers are reasonable approaches.

Layer 2/3/4 Load balancers operating on layer 3/4 are either *web switches*—dedicated, proprietary network appliances ("black-boxes")—or *IP virtual servers* operating on commodity server hardware. Their functionality resembles a reverse NAT mapping or routing. Instead of mapping Internet access for multiple private nodes via a single IP, they provide a single externally accessible IP mapped to a bunch of private servers. Layer 2 balancers use link aggregations and merge multiple servers to a single logical link. All these approaches use mechanisms such as transparent header modifications, tunneling, switching or routing, but on different layers. Dedicated network appliances can provide impressive performances in terms of throughput, alas with a heavy price tag. Solutions based on *IP virtual servers* running on regular hardware often provide a more affordable solution with reasonable performance up to a certain scale.

Layer 7 Load balancers operating on the application layer are essentially reverse proxies in terms of HTTP. As opposed to the balancers working on lower layers, the layer 7 load balancers can take advantage of explicit protocol knowledge. This comes with clear performance penalties due to a higher overhead of parsing traffic up to the application layer. Benefits of this technique are the the possibility of HTTP-aware balancing decisions, potential caching support, transparent SSL termination and other HTTP-specific features. Similar to IP virtual servers, layer 7 balancers are less performant than web switches. But being hosted on commodity hardware results in a decent horizontal scalability.

Balancing Strategies

Various load-balancing strategies have been developed [Sch06, Sch08] and the design of effective balancing algorithms is still a matter of academic interest in the era of Cloud Computing [Ran10]. A major challenge of strategies is the difficulty to anticipate future requests. The missing a priori knowledge limits the strategies to make only a few assumptions based on the recent load, if any assumptions are made at all.

Round Robin All servers are placed in a conceptual ring which gets rotated on each request. A drawback of this simple mechanism is the unawareness of overloaded servers.

Least Connections Following this strategy, the balancer manages a list of servers and their active connection count. New connections are forwarded based on this knowledge. The idea rests on the fact that connections seize machine resources and the machine with the least connections or the smallest backlog has still the most capacities available.

Least Response Time This strategy is similar to the previous one, but uses the response times instead of connection counts. The rationale behind the metering via response times is based on the expressiveness of latencies for the server load, especially when the workloads per connection differ.

Randomized In a randomized strategy, the load balancers picks a backend web server by chance. This mechanism achieves suprisingly good results, thanks to probabilistic distribution.

Resource-aware A resource-aware strategy utilizes external knowledge about the servers' utilizations, but also metrics suchs as connection counts and response times. The values are then combined to weight all servers and distribute load correspondingly.

Besides these basic strategies, there are various advanced algorithms that often combine different approaches. Furthermore, load-balancing strategies become more difficult, when there are more than one load balancers deployed at the same time. Some of the strategies estimate utilization based on their forwarding decisions. Multiple load balancers might interfere the individual assumptions. As a result, cooperative strategies are often required that share knowledge between balancers.

According to Schlossnagle [Sch06], a 70% per-server utilization is a respectable goal in a larger architecture. Higher utilizations are unrealistic due to short-liveness of tasks, the high throughput rate and missing future knowledge about incoming requests.

Session Stickiness

Session stickiness is a technique to map a certain user accessing a web application to the same backend web server during his browsing session. Hence, it is sufficient to store session states on the respective servers. While session stickiness is inherently provided in single server setups,

this concept is very difficult when it comes to load balancing. Essentially, session stickiness requires the load balancer to forward a request according to the session used (i.e. parsing a cookie, reading a session variable or customized URI). As a result, the load balancer starts to distribute sessions to machines instead of single connections and requests. While this setup is attracting and handy for web application developers, it represents are severe challenge from a scalability and availability perspective. Loosing a server equates a loss of associated sessions. The granularity of single requests enables a sound distribution mechanism when demand increases. New servers can be added to the architecture, but splitting and reallocating existing sessions already bound to distinct machines is complex. As a result, the concepts of effective resource utilization and allocation of session data should not be conflated, otherwise scalability and availability are in danger. In terms of load balancing, session stickiness is regarded as a misconception and should be avoided [Sch06]. Instead, a web architecture should provide means to access session data from different servers. A more stateless communication, where the clients manage session state, further mitigates the problem of session stickiness.

3.2 Cloud Architectures

Within the last years, the increasing presence of large-scale architectures has introduced the era of Cloud Computing.

3.2.1 Cloud Computing

There is a huge number of different definitions available towards Cloud Computing. The term "cloud" itself is derived from the figurative abstraction of the internet represented as a cloud. The available definitions of Cloud Computing range from reductions it to be "the next hype term" over pragmatic definitions focusing on special aspects to definitions, that see Cloud Computing as a general paradigm shift of information architectures, as an *ACM CTO Roundtable* [Cre09] has shown in 2009. Of the various features that are attributed to the "cloud", scalability, a pay-per-use utility model and virtualization are the minimum common denominators.

The key concept of Cloud Computing is the resourcing of services. These services can be generally differentiated into three vertical layers:

Infrastructure as a Service The provision of virtualized hardware, in most cases as virtual computing instances, is termed Infrastructure as a Service (IaaS). The particular feature of this form of cloud service is the ability to change these instances on demand and at any time. This includes spawning new computing instances, altering existing ones by resizing or reassigning resources or shutting down unneeded instances dynamically. Basic working units for IaaS are virtual images, that are instantiated and deployed in the cloud and executed within virtual machines. They contain an operating system and additional software

implementing the application. Due to complete virtualization, the physical infrastructure of the data centers of IaaS providers is totally transparent to their customers.

Platform as a Service Platform as a Service (PaaS) provides an additional level of abstraction by offering an entire runtime environment for cloud-based applications. PaaS typically supplies a software stack of dedicated components where applications can be executed on, but also tools facilitating the development process. The platform hides all scalability efforts from the developer, thus it appears as one single system to the user, although it transparently adapts to scale. Most of the platforms realize this feature through elaborate monitoring and metering. Once an application is uploaded onto the platform, the system automatically deploys it within the nodes of the data center of the PaaS. When the application is running under heavy load, new nodes will be added automatically for execution. If demand declines, spare nodes will be detached from the application and returned into the pool of available resources.

Software as a Service Software as a Service (SaaS) provides a web-centric supply of applications. Due to various technological trends, web-based applications mature and become powerful pieces of software running entirely within the browser of the user. By replacing traditional desktop applications that run locally, SaaS providers are able to publish their applications solely in the web. Cloud architectures allow them to cope with a huge number of users online. It allows users to access the applications on demand and they can be charged merely by usage without having to buy any products or licenses. This reflects the idea to consider the provision of software *as a service*.

Referring to the terminology of Cloud Computing, a scalable web application is essentially a SaaS that requires appropriate execution/hosting environments (PaaS and/or IaaS).

3.2.2 PaaS and IaaS Providers

In the following, we will consider some exemplary hosting providers and outline their service features.

Amazon Web Services

Amazon was one of the first providers for dedicated, on-demand and pay-per-use web services[1]. It is currently dominating the multi-tenant Cloud Computing market. Their main product is *Elastic Compute Cloud (EC2)*, a service providing different virtualized private servers. The provision of virtualized machines in scalable amounts forms an architectural basis for most of their clients. Instead of growing and maintaining own infrastructure, EC2 clients can spin up new machine

1 `http://aws.amazon.com/`

instances within seconds and thus cope with varying demand. This traditional IaaS hosting is complemented by a set of other scalable services that represent typical architecture components.

Elastic Block Storage (EBS) provides block-level storage volumes for EC2 instances. *Simple Storage Service (S3)* is a key-value web-based storage for files. More sophisticated data storage systems available are *SimpleDB*, *DynamoDB* and *Relational Database Service (RDS)*. The former two services represent non-relational database management systems with a limited set of features. RDS currently supports MySQL-based and Oracle-based relational databases.

Elastic Load Balancing (ELB) provides load-balancing functionalities on transport protocol level (e.g. Transmission Control Protocol (TCP)) and application level (e.g. HTTP). As a message queue, *Simple Queue Service (SQS)* can be used. For complex MapReduce-based computations, Amazon has introduced *Elastic MapReduce*.

ElastiCache is an in-memory cache system that helps speeding up web applications. *CloudFront* is a Content Delivery Network (CDN), complementing S3 by replicating items geographically. For monitoring purposes, Amazon has come up with *CloudWatch*, a central real-time monitoring web service.

Besides these services, Amazon has introduced their own PaaS stack, called *Elastic Beanstalk*. It is essentially a bundle of existing services such as EC2, S3 and ELB and allows to deploy and scale Java-based web applications. Furthermore, there are additional services that cover business services such as accounting or billing.

Google App Engine

The Google App Engine[1] is a PaaS environment for web applications. It currently provides support for Python, Go and several JVM-based languages such as Java. Applications are hosted and executed in data centers managed by Google. One of its main features is the automatic scaling of the application. The deployment of the application is transparent for the user, and applications encountering high load will be automatically deployed to additional machines.

Google provides free usage quotas for the App Engine, limiting traffic, bandwidth, number of internal service calls and storage size among others. After exceeding these limits, users can decide to add billable resources and are thus getting charged for additional capacities.

The runtime environment of the application is sandboxed and several language features are restricted. For instance, JVM-based applications cannot spawn new threads, use socket connections and access the local file system. Furthermore, the execution of a request must not exceed a 30 seconds limit. This restrictions are enforced by modified JVMs and altered class libraries.

Besides an application container, the App Engine provides several services and components as part of the platform. They are accessible through a set of APIs. We will now have a brief look at the current Java API.

1 `https://appengine.google.com/`

The *Datastore API* is the basic storage backend of the App Engine. It is schemaless object datastore, with support for querying and atomic transaction execution. Developers can access the datastore using Java Data Objects, Java Persistence API interfaces or via a special low-level API. A dedicated API provides support for asynchronous access to the datastore. For larger objects, and especially binary resources, the App Engine provides a *Blobstore API*. For caching purposes, the *Memcache API* can be used. Either using a low-level API or the JCache interface.

The *Capabilities API* enables programmatical access to scheduled service downtimes and can be used to develop applications that prepare for the unavailablity of capabilities automatically. The *Multitenancy API* provides support for multiple separated instances of an application running in the App Engine.

Images can be manipulated using the dedicated *Images API*. The *Mail API* allows the application to send and receive emails. Similarly, the *XMPP API* allows message exchange based on Extensible Messaging and Presence Protocol (XMPP). The *Channel API* can be used to establish high-level channels with clients over HTTP and then push messages to them.

The *Remote API* opens an App Engine application for programmable access using an external Java application. Other web resources can be accessed using the *URLFetch API*.

Thanks to the *Task Queue API*, there is some support for tasks decoupled from request handling. Requests can add tasks to a queue, and workers asynchronously execute the background work. The *Users API* provides several means to authenticate users, including Google Accounts and OpenID identifiers.

3.3 An Architectural Model for Scalabale Web Infrastructures

The need for scalable web architectures is much older than the set of concepts that is subsumed as cloud computing. Pioneer web corporations have learned their own lessons when interest in their services gradually increased and existing capacities where exploited. They were forced to design and build durable infrastructures that were able to keep up with increasing demands. We have a look at some of the more important guidelines and requirements in the following. Then we introduce an architectural model for scalable web infrastructures that is derived from existing cloud infrastructures and addresses the requirements. The resulting model will provide the conceptual basis for our further considerations and allows us to survey concurrency in specific components in the following chapters.

3.3.1 Design Guidelines and Requirements

Let us first make some assumptions on the general design of our infrastructure. We are targeting scalability, thus we need to provide a proper scalability path. Although vertical scalability can help us here, it is not our major vehicle for scaling. When replacing a single-core CPU with a quad-core machine, we may quadruple the overall performance of that node (if at all). However, we will quickly hit technical constraints and especially limitations of cost effectiveness when following

this path to scale up further. Thus, our primary mean of growing is horizontal scaling. In an ideal situation, this enables us to scale our infrastructural capacities linearly with the provision of additional nodes. Horizontal scale inherently forces some kind of distribution. Being much more complex than regular systems, distributed systems introduce a lot of additional problems and design issues to keep in mind (see subsection 2.3.5). Above all, this is the acceptance and graceful handling of failures within the system. These two requirements form our fundamental design basis for the infrastructure: We must design for scale, and we must design for failure. It is tremendously hard to scale a running system that has not been designed for it. It is neither easy to fix a complex system that fails due to partial failures of conflated subcomponents. As a result, we will now introduce some guidelines to build a scalable infrastructure from ground up.

When designing for scale, we are targeting horizontal scalability. Thus, system partitioning and component distribution are essential design decisions right at the start. We begin with allocating the components into loosely coupled units and overplanning of capacities and resources. Decoupling of components prevents the (accidental) development of an irreducibly conflated, and highly complex system. Instead, a decoupled architecture suggests a simplified model that eases capacity planning, and induces less coherency requirements between components. Isolating components and subservices allows to scale them independently and enables the system architect to apply different scaling techniques for the components, such as cloning, splitting and replicating. This approach prevents overengineering a monolithic system and favors simpler and easier components. Abbott et al. [Abb11] suggest to deliberately overplan capacities during design, implementation and deployment. As a ballpark figure, they recommend factor $20x$ during the design phase, factor $3x$ for the implementation and at least factor $1.5x$ for the actual deployment of the system.

Designing for failure has several impacts on the architecture. An obvious statement is to avoid single points of failure. Concepts like replication and cloning help to build redundant components that tolerate the failure of single nodes without compromising availability. From a distribution perspective, Abbott et al. [Abb11] further recommend a breakdown into *failure domains*. These are domains that group components and services in a way that a failure within does not affect or escalate to other failure domains. Failure domains help both to detect and isolate partial faults in the system. The concept of so-called *fault isolative swim lanes* takes this idea to the extreme and disallows synchronous calls between the domains entirely and also discourages asynchronous calls. Sharding the complete architecture for different groups of users is an example of fault isolative swim lanes.

Next, we need a way to communicate between components. It is common practice ([Abb09, Hel12]) to use messaging since it provides an adequate communication mechanism for integrating loosely coupled components. Alternatives such as RPC are less applicable because they require stronger coupling and coherency between components.

Another set of advices for scalable architectures concerns time and state, fundamental challenges of distributed systems. As a rule of thumb, all temporal constraints of a system should be relaxed as far as practicable and global state should be avoided whenever possible. Of course, the

applicability of these advices depends on the actual use case blatantly. For instance, a large-scale online banking system cannot even temporarily tolerate invalid account balances. On the other hand, many applications describe use cases that allow to weaken temporal constraints and consistency to some extent. When a user of social network uploads an image or writes a comment, there is no degradation of service when it takes some seconds until it eventually appears for other users. If there are means for relaxing temporal constraints, using asynchronous semantics and preventing distributed global state, they should be considered in any case. Enforcing synchronous behavior and coordinating shared state between multiple nodes are among the hardest problems of distributed systems. Bypassing these challenges whenever possible greatly increases scalability prospects. These advices have not only implications for implementations, but also for the general architecture. Asynchronous behavior and messaging should be used unless there is a good reason for synchronous semantics. Weakening global state helps to prevent single point of failures and facilitates service replication.

Caching is another mechanism that helps to provide scalability at various spots. Basically, caching is about storing copies of data with higher demand closer to the places it is actually needed inside the application. Caching can also prevent multiple executions of an idempotent operation by storing and reusing the result. Components can either provide their own internal caches. Alternatively, dedicated caching components provide caching functionalities as a service to other components of the architecture. Although caching can speed up an application and improve scalability, it is important to reason on appropriate algorithms for replacement, coherence and invalidation of cached entries.

In practice, it is very important to incorporate logging, monitoring and measuring facilities into the system. Without detailed data about utilization and load, it is difficult to anticipate increasing demand and detect bottlenecks before they become critical. Taking countermeasures such as adding new nodes and deploying additional component units should be always grounded on a comprehensive set of rules and actual measurements.

3.3.2 Components

Following these guidelines, we now introduce an architectural model for scalabale web infrastructures. It is based on separated components, that provide dedicated services and scale independently. We group them into layers with common functionalities. The components are loosely coupled and use messaging for communication.

For each class of component, we describe its task and purpose and outline a suitable *scalability strategy*. Also, we name some *real-world examples* that fit into the particular class and refer to *cloud-based examples* from the popular platforms mentioned before.

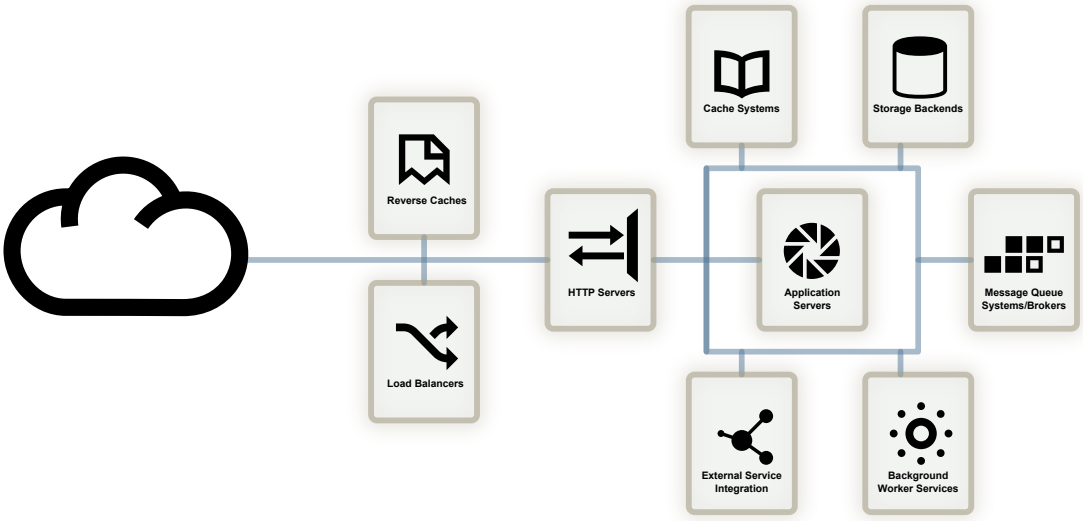

Figure 3.1: An architectural model of a web infrastructure that is designed to scale. Components are loosely coupled and can be scaled independently.

HTTP Server

The first component of our model is the actual HTTP server. It is a network server responsible for accepting and handling incoming HTTP requests and returning responses. The HTTP server is decoupled from the application server. That is the component that handles the real business logic of a request. Decoupling both components has several advantages, primarily the separation of HTTP connection handling and execution of application logic . This allows the HTTP server to apply specific functions such as persistent connection handling, transparent SSL/TLS encryption[1] or on-the-fly compression of content without impact on the application server performance. It also decouples connections and requests. If a client uses a persistent connection issuing multiple requests to a web server, the requests can still be separated in front of the application server(s). Similarly, the separation allows to scale both components independently, based on their individual requirements. For instance, a web application with a high ratio of mobile users has to deal with many slow connections and high latencies. The mobile links cause slow data transfers and thus congest the server, effectively slowing down its capability of serving additional clients. By separating application servers and HTTP servers, we can deploy additional HTTP servers upstream and gracefully handle the situation by offloading the application server.

Decoupling HTTP servers and application servers requires some kind of routing mechanism

1 Applicability depends on load-balancing mechanism used.

that forwards requests from a web server to an application server. Such a mechanism can be a transparent feature of the messaging component between both server types. Alternatively, web servers can employ allocation strategies similiar to the strategies of load balancers (cf. subsection 3.1.3). Another task for some of the web servers is the provision of static assets such as images and stylesheets. Here, local or distributed file systems are used for content, instead of dynamically generated contents provided by the application servers.

Scalability strategy: The basic solution for HTTP servers is cloning. As they do not hold any state in our case, this is straightforward.
Real world examples: The Apache HTTP Server[1] is currently the most popular web server, although there is an increasing number of alternatives that provides better scalability, such as nginx[2] and lighttpd[3].
Cloud-based examples: The Google App Engine internally uses Jetty[4], a Java-based, high-performance web server.

The implications of scalability and concurrency in case of huge amounts of parallel connections and requests is the main topic of chapter 4.

Application Server

The application server is a dedicated component for handling requests at application level. An incoming request, which is usually received as preprocessed, higher-level structure, triggers business logic operations. As a result, an appropriate response is generated and returned.

The application server backs the HTTP server and is a central component of the architecture, as it is the only component that incorporates most of the other services for providing its logic. Exemplary tasks of an application server include parameter validation, database querying, communication with backend systems and template rendering.

Scalability strategy: Ideally, application servers adhere to a *shared nothing* style, which means that application servers do not share any platform resources directly, expect for a shared database. A shared nothing style makes each node independent and allows to scale via cloning. If necessary, coordination and communication between application servers should be outsourced to shared backend services. If there is a tighter coupling of application server instances due to inherent shared state, scalability becomes very difficult and complex at a certain scale.
Real world examples: Popular environments for web application include dedicated scripting

1 http://httpd.apache.org/
2 http://nginx.org/
3 http://www.lighttpd.net/
4 http://jetty.codehaus.org/jetty/

languages such as Ruby (on Rails), PHP or Python. In the Java world, application containers like RedHat's JBoss Application Server or Oracle's GlassFish are also very popular.
Cloud-based examples: The Google App Engine as well as Amazon's Elastic Beanstalk support the Java Servlet technology. The App Engine can also host Python-based and Go-based applications.

Programming and handling concurrency inside an application server is subject of chapter 5.

Load Balancer & Reverse Cache

So far, our outer component was the HTTP server. For scaling via multiple web servers, we need an upstream component that balances the load and distributes incoming requests to one of the instances. Such a load balancer works either on layer 3/4 or on the application layer (layer 7).

An application layer load balancer represents a reverse proxy in terms of HTTP. Different load balancing strategies can be applied, such as round-robin balancing, random balancing or load-aware balancing (also see subsection 3.1.3). A reverse cache can further improve the performance and scalability by caching dynamic content generated by the web application. This component is again an HTTP reverse proxy and uses caching strategies for storing often requested resources. Thus, the reverse proxy can directly return the resource without requesting the HTTP server or application server.

In practice, load balancers and reverse caches can both appear in front of a web server layer. They can also be used together, sometimes even as the same component.

Scalability strategy: Load balancers can be cloned as well. However, different strategies are required to balance their load again. A popular approach to balance a web application is to provide multiple servers to a single hostname via DNS. Reverse caches can be easily cloned, as they provide an easy parallelizable service.
Real world examples: There are several products used by large-scale web applications. Popular load balancers are HAProxy[1], perlbal[2] and nginx. Reverse proxies with dedicated caching functionalities include Varnish[3] and again nginx.
Cloud-based examples: Amazon provides a dedicated load-balancing service, ELB.

Some of the considerations of chapter 4 are also valid for load balancers and reverse proxies.

Message Queue System

Some components require special forms of communication, such as HTTP-based interfaces (e.g. web services) or low-level socket-based access (e.g. database connections). For all other

1 http://haproxy.1wt.eu/
2 http://danga.com/perlbal/
3 https://www.varnish-cache.org/

communication between components of the architecture, a message queue system or message bus is the primary integration infrastructure. Messaging systems may either have a central message broker, or work totally decentralized. Messaging systems provide different communication patterns between components, such as request-reply, one-way, publish-subscribe or push-pull (fan-out/fan-in), different synchronicity semantics and different degrees of reliability.

Scalability strategy: A decentralized infrastructure can provide better scalability, when it has no single point of failure and is designed for large deployments. Message-oriented middleware systems with a message broker require more sophisticated scaling approaches. These may include partitioning of messaging participants and replication of message brokers.
Real world examples: Advanced Message Queuing Protocol (AMQP) [AMQ11] is a popular messaging protocol with several mature implementations, such as RabbitMQ[1] . A popular broker-free and decentralized messaging system is øMQ[2].
Cloud-based examples: Amazon provides a SQS, a message queueing solution. The Google App Engine provides a queue-based solution for the handling of background tasks and a dedicated XMPP messaging service. However, all of these services have higher latencies (up to several seconds) for message delivery, are thus designed for other purposes. It is not reasonable to use these services as part of HTTP request handling, as such latencies are not acceptable. However, several EC2-based custom architectures have rolled out their own messaging infrastructure, based on the aforementioned products such as øMQ.

Backend Data Storage

These components allow to store structured, unstructured and binary data as well as files in a persistent, durable way. Depending on the type of data, this includes relational database management systems, non-relational database management systems, and distributed file systems.

Scalability strategy: Scaling data storages is a challenging task, as we will learn in chapter 6. Replication, data partitioning (i.e. denormalization, vertical partitioning) and sharding (horizontal partitioning) represent traditional approaches.
Real world examples: MySQL[3] is a popular relational database management system with clustering support. Riak[4], Cassandra[5] and HBase[6] are typical representatives of scalable, non-relational

1 http://www.rabbitmq.com/
2 http://www.zeromq.org/
3 https://www.varnish-cache.org/
4 https://github.com/basho/riak
5 http://cassandra.apache.org/
6 http://hbase.apache.org/

database management systems. HDFS[1], GlusterFS[2] and MogileFS[3] are more prominent examples for distributed file systems used inside large-scale web architectures.
Cloud-based examples: The Google App Egnine provides both a data store and a blob store. Amazon has come up with several different solutions for cloud-based data storage (e.g. RDS, DynamoDB, SimpleDB) and file storage (e.g. S3).

Chapter 6 addresses the challenges of concurrency and scalability of storage systems.

Cache System

In contrast to durable storage components, caching components provide a volatile storage. Caching enables low-latency access to objects with high demand. In practice, these components are often key/value-based and in-memory storages, designed to run on multiple nodes. Some caches also support advanced features such as publish/subscribe mechanisms for certain keys.

Scalability strategy: Essentially, a distributed cache is a memory-based key/value store. Thus, vertical scale can be achieved by provisioning more RAM to the machine. A more sustainable scale is possible by cloning and replicating nodes and partitioning the key space.
Real world examples: Memcached[4] is a popular distributed cache. Redis[5], another in-memory cache, supports structured data types and publish/subscribe channels.
Cloud-based examples: The Google App Engine supports a Memcache API and Amazon provides a dedicated caching solution called ElastiCache.

Some of the considerations of chapter 6 are also valid for distributed caching systems.

Background Worker Service

Computationally-intensive tasks should not be executed by the application server component. For instance, transcoding uploaded video files, generating thumbnails of images, processing streams of user data or running recommendation engines belong to these CPU-bound, resource-intensive tasks. Often, these tasks are asynchronous, which allows them to be executed in the background independently.

1 http://hadoop.apache.org/hdfs/
2 http://www.gluster.org/
3 http://danga.com/mogilefs/
4 http://memcached.org/
5 http://redis.io/

Scalability strategy: Adding more resources and nodes to the background worker pool generally speeds up the computation or allows the execution of more concurrent tasks, thanks to parallelism. From a concurrency perspective, it is easier to scale worker pools when the jobs are small, isolated tasks with no dependencies.
Real world examples: Hadoop[1] is an open-source implementation of the MapReduce platform, that allows the parallel execution of certain algorithms on large data sets. For real-time event processing, Twitter has released the Storm engine[2], a distributed realtime computation system targeting stream processing among others. Spark [Zah10] is an open source framework for data analytics designed for in-memory clusters.
Cloud-based examples: The Google App Engine provides a Task Queue API, that allows to submit tasks to a set of background workers. Amazon offers a custom MapReduce-based service, called Elastic MapReduce.

Integration of External Services

Especially in an enterprise environment, it is often required to integrate additional backend systems, such as CRM/ERP systems or process engines. This is addressed by dedicated integration components, so-called enterprise service buses. An ESB may also be part of a web architecture or even replace the simpler messaging component for integration. On the other hand, the backend enterprise architecture is often decoupled from the web architecture, and web services are used for communication instead. Web services can also be used to access external services such as validation services for credit cards.

Scalability strategy: The scalability of external services depends first and foremost on their own design and implementation. We will not consider this further, as it is not part of our internal web architecture. Concerning the integration into our web architecture, it is helpful to focus on stateless and scalable communication patterns and loose coupling.
Real world examples: Mule[3] and Apache ServiceMix[4] are two open-source products providing ESB and integration features.
Cloud-based examples: Both of the providers we regarded make integration mechanisms for external services available only on a lower level. The Google App Engine allows to access external web-based resources via URLFetch API. Furthermore, the XMPP API may be used for message-based communication. Similarly, the messaging services from Amazon may be used for integration.

1 http://hadoop.apache.org/
2 https://github.com/nathanmarz/storm
3 http://www.mulesoft.org/
4 http://servicemix.apache.org/

3.3.3 Critical Reflection of the Model

We have now introduced a general architectural model of a web infrastructure that has the capabilities to scale. However, the suggested model is not entirely complete for direct implementation and deployment. We neglected components that are not necessary for our concurrency considerations, but that are truly required for operational success. This includes components for logging, monitoring, deployment and administration of the complete architecture. Also, we omitted components necessary for security, and authentication due to simplicity. When building real infrastructures, it is important to also incorporate these components.

Another point of criticism targets the deliberate split of components. In practice, functionality may be allocated differently, often resulting in fewer components. Especially the division of web server and application server may seem arbitrary when regarding certain server applications in use today. However, a conflated design affects our concurrency considerations and often states a more difficult problem in terms of scalability.

3.4 Scaling Web Applications

We have regarded the scalability of web applications from an architectural point of view so far. In the next chapters, we will focus on scalability within web architectures, based on the usage of concurrency inside web servers, application servers and backend storage systems.

However, there are other factors which influence the scalability and perceived performance of web applications. Therefore, we will provide a brief overview of factors relevant for web site setup and client-side web application design. The overview summarizes important strategies outlined in relevant books[All10, Abb11, Abb09] and a blog article from the Yahoo Developer Network[1].

From a user's perspective, a web applications appears scalable, when it continues to provide the same service and the same quality of service independent of the number of concurrent users and load. Ideally, a user should not be able to draw any inferences from his experience interacting with the application about the actual service load. That is why constant, low-latency responses are important for the user experience. In practice, low round-trip latencies of single request/response cycles are essential. More complex web applications mitigate negative impacts by preventing full reloads and through dynamic behaviour, such as asynchronous loading of new content and partial update of the user interface (c.f. AJAX). Also, sound and reliable application functions and graceful degradation are crucial for user acceptance.

3.4.1 Optimizing Communication and Content Delivery

First and foremost, it is very important to minimize the number of HTTP requests. Network round trip times, perhaps preceded by connection establishing penalties, can dramatically increase

1 `http://developer.yahoo.com/performance/rules.html`

latencies and slow down the user experience. Thus, as few requests as necessary should be used in a web application. A popular approach is the use of CSS sprites or images maps[1]. Both approaches load a single image file containing multiple smaller images. The client then segments the image and the smaller tiles can be used and rendered independently. This technique allows to provide a single image containing all button images and graphical user interfaces elements as part of a combined image. Another way of reducing requests is the inlining of external content. For instance, the data URI scheme [Mas98] allows to embed arbitrary content as a base64-encoded URI string. In this way, smaller images (e.g. icons) can be directly inlined into an HTML document. Browsers often limit the number of parallel connections to a certain host, as requested in RFC 2616 [Fie99]. So when multiple resources have to be accessed, it can help to provide several domain names for the same server (e.g. static1.example.com, static2.example.com). Thus, resources can be identified using different domain names and clients can increase the number of parallel connections when loading resources.

Another important strategy is to embrace caching whenever possible. As a rule of thumb static assets should be indicated as not expiring. Static assets are images, stylesheet files, JavaScript files and other static resources used to complement the site. These assets are generally not edited anymore, once online. Instead, they get replaced by other assets when the site gets updated. This yields some immutable character for static assets, that allows us to cache aggressively. Consequently, static asset caching narrows the traffic between web browsers and web servers almost exclusively to dynamically generated content after some time. For dynamic content, web servers should provide useful headers (e.g. `ETag`, `Last-Modified`) allowing conditional semantics in subsequent requests. Reverse caching proxies as part of the web architecture can also cache generated content and speed up responses.

The use of CDNs helps to off-load the machines serving static assets and shortens response times. As CDNs are not just caching reverse proxies, they also provide geographical distribution and route requests to the nodes in the closest proximity of the client. Using public repositories for popular assets such as JavaScript libraries (e.g. jQuery) is also beneficial. These repositories are not just backed by large CDNs, their use also increases the chance of being already cached.

On-the-fly compression of content can further reduce size with marginal CPU overhead. This is helpful for text-based formats and especially efficient in case of formats with verbose syntax (e.g. HTML, XML).

3.4.2 Speeding up Web Site Performance

There are also some advice for the structure of HTML files, improving the user experience. It is preferable to reference external stylesheets at the top of the HTML file, and reference JavaScript files at the bottom. When parsing and rendering the site, the browser will first load stylesheets and then the scripts, depending on the number of parallel requests allowed (see above). This order

1 `http://www.alistapart.com/articles/sprites`

helps to render the page progressively and then include interactive elements. Also, JavaScript and CSS resources should not be inlined in the HTML file, but externalized to files. This helps caching efforts and minimizes the HTML file size. Text-based assets like stylesheets and JavaScript files should also be textually minified by removing whitespaces and renaming variable names to shortest names possible, effectively reducing file sizes in production environments. Different timings of content loading can also affect the user experience. Rich user interfaces relying on AJAX-based content often provide a basic user interface on the first load. After that, actual content is accessed and filled into the user interface. This order results in a gradual site composition, that appears to be more responsive than larger HTML files with complete reloads. Opposed to this post-loading of content, also preloading techniques can speed up the user experience. HTML5 provides a `prefetch` link relation, allowing the browser to load the linked resources in advance. As rich, web-based interfaces increasingly use the browser for application features and parts of the business logic, it is also important to handle computations and concurrent tasks efficiently. This has been addressed with the Web Worker API [Hic09c], that allows to spawn background workers. For communication and coordination, workers use message-passing, they do not share any data.

Concerning the application, it is helpful to have a clear separation of different states. Ideally, the server handles the persistent states of application resources, while the clients handle their own session states and communication is stateless. Traditionally, entirely stateless communication has been difficult, as HTTP cookies have been the primary way of stateful web applications. The Web Storage API [Hic09b], related to HTML5, provides an alternative mechanism for client-side state handling. This API essentially allows to store key/value pairs via JavaScript in the browser. The API provides persistent state per domain, or scoped to a single browser session. As opposed to HTTP cookies, Web Storage API operations are not placed into HTTP headers, hence they keep the communication stateless. This browser storage can also be used to cache data inside the client-side application layer.

3.5 Summary

We have seen that delivering dynamic content represents a different challenge than serving static files. Techniques like CGI, FastCGI and dedicated web server modules have evolved to integrate dynamic content, and scripting languages are very popular in this context. Alternatively, containers that host dynamic web applications can be used for deployment, which is the preferred way for Java-based applications. A tiered view on web architectures not merely helps to split up deployment-specific components. It can also provide a logical separation of concerns by layering presentation, application logic and persistence. We also learned that load balancing is a fundamental concept for scalability in web architectures, as it enables the distribution of load to multiple servers.

We then introduced the general principles of Cloud Computing, the resourcing of services

with a pay-per-use utility model and on demand, using virtualization techniques. Being geared towards seamless scalability, Cloud Computing platforms and infrastructures relate well to our endeavor of scalable web architectures.

Based on similar components from existing Cloud services, we have introduced an architectural model for scalabale web infrastructures. Instead of a single, monolithic architecture, our model adheres to a set of dedicated service components with loose coupling. The separation allows us to scale different service components using appropriate scaling strategies and makes the overall architecture more robust, reliable and agile. The isolation of web servers, applications servers and backend storage systems also enables us to benefit from a more precise view on concurrency challenges inside each component type.

Other attempts at improving performance and scalability have been mentioned, aiming at optimized communication, better content delivery and faster web sites. However, they are not part of further considerations in the subsequent chapters.

4 Web Server Architectures for High Concurrency

In this chapter, we have a closer look at concurrency when handling multiple connections inside a web server. As outlined in our architectural model, request handling is decoupled from application logic. Thus, the creation of dynamic content is out of scope here, and will be investigated separately in the subsequent chapter 5. From a web server perspective, there is also no coordination required between two distinct requests. If there are dependencies between requests on the application level—e.g. a long-polling request waiting for application events and another request triggering an application event—they must be handled inside the application server in our model.

This chapter features an overview on the challenges of concurrency for high-performance web servers with huge numbers of simultaneous connections. Next, we learn how different server architectures handle parallelism, using different programming models for concurrency and I/O operations. We finally take a step back and compare the prevalent concurrency models—thread-based and event-driven—more generally.

4.1 Overview

The main issue we address in this chapter is the appropriate mapping of connections/requests to concurrent flows of executions in a programming model. As we are targeting multiple parallel HTTP requests, this mainly involves highly I/O-bound operations. Concerning web infrastructures, we want to make sure that our software implementation does not easily become the bottleneck and a high utilization of hardware resources is achieved under load on each deployed server. In terms of request/response and connection handling, there are several interesting metrics for describing a server's performance:

- Request throughput (#/sec)
- Raw data throughput (Mbps)
- Response times (ms)
- Number of concurrent connections (#)

Furthermore, there are the following performance statistics to be observed locally on the server's machine:

- CPU utilization
- Memory usage
- Number of open socket/file handles
- Number of threads/processes

To restate our problem, we want to handle as many requests in parallel as possible, as fast as possible and with as few resources as necessary. In other words, the resource utilization is to be scaling with increasing work.

4.1.1 Request Handling Workflow

Based on the web server requirements we outlined in our previous chapter, we can list the following steps as the minimal workflow for handling a request. Additional features such as request logging or caching are left out on purpose.

1. *Accept the incoming request* – In case of a new HTTP connection, an underlying TCP connection must be established first.

2. *Read the request* – Reading the raw bytes from the socket (I/O-bound) and then parsing the actual HTTP request (CPU-bound) is necessary for all requests. If a request contains an entity, such as POST parameters or a file upload, this additional content must be read as well. Depending on the implementation, the web server either buffers the entity until loaded completely or pipes it directly to the application server. The former allows content offloading and is important for slow connections, the latter is interesting because of decreased latencies.

3. *Dispatch the request tp the application level* – Based on our architectural model, the parsed request is then issued to the application server. We use decoupled components, so this is generally a network-based task, using messaging (or alternatives such as RPC). In case of a web server handling static content, we access a local or remote file system instead. All operations are I/O-bound.

4. *Write the generated response to the socket, once available* – Once the response is generated (e.g. a generated HTML file from the application server or a static image from the file system), it can be returned to the client by writing to the socket. Again, the web server can either buffer the response and thus provide offloading for the application servers. Or it pipes the generated response directly to the client.

5. *Finish the request* – Depending on the connection state negotiated via the request/response headers and the HTTP defaults, the web server either closes the connection, or it starts from the beginning and awaits the next request to be send from the client.

4.1.2 The C10K Problem

Kegel has published a seminal article [Keg06] in 1999 on this problem, proclaiming that "it is time for web servers to handle ten thousand clients simultaneously", hence coining the term of the *C10k problem*. The original article was updated several times and became an influential resource on web server scalability.

He motivates his considerations by showing that hardware might no longer be the bottleneck for high connection concurrency to a certain extent. Based on reasonable hardware at that time (i.e. 500 MHz, 1 GB of RAM, 6 x 100Mbit/s), Kegel argues that 10.000 parallel clients are totally feasible, yielding 50KHz, 100Kbytes, and 60Kbits/sec per request – quite enough for 4kb of payload data. In practice, most servers were far away from that number at that time. He then examined web server internals and evaluated common I/O strategies and threading models in particular.

The C10k term has been reinforced ten years later, when the the company *Urban Airship* struggled to serve 500.000 concurrent connections on a single node[1]. Their interest in solving the *C500k problem* was based on their business model. Providing notification services to huge numbers of mobile devices requires them to handle extraordinary high numbers of idle connections in parallel.

4.1.3 I/O Operation Models

For regular desktop applications, handling file-based or network-based input and output is often a sporadic task. For our web servers, it is the primary task to handle I/O operations. Operating systems provide different means for I/O operations, and we will now have a closer look at I/O operation models.

The terms blocking and synchronous resp. non-blocking and asynchronous are often used exchangeable in the literature and both describe very similar concepts. Also, the terms are used on different levels on different operating systems with different meanings. We separate them at least for the description of I/O operations.

blocking vs. non-blocking Using these properties, the application can tell the operating system how to access the device. When a blocking mode is used, the I/O operation does not return to the caller unless the operation has finished. In a non-blocking mode, all calls return immediately, but only indicate the call status of the operation or the errors. Thus, multiple calls may be required to await the successful end of the operation.

1 http://urbanairship.com/blog/2010/08/24/c500k-in-action-at-urban-airship/

synchronous vs. asynchronous These properties are used to describe the control flow during the I/O operation. A synchronous call keeps control in the sense of not returning unless the operation has finished. An asynchronous call immediately returns, allowing to execute further operations.

Combining these modes yields four distinct operation models for I/O. Sometimes, an additional software layer is used to provide a different model than the actual underlying model for convenience.

synchronous blocking I/O This is the most common operational mode for many regular applications. In this case, an I/O operation is a single operation, resulting in a blocked application state until the operation has completed and data has been copied from kernel space to user space (i.e. read operation). On kernel level, the actual raw operation is often multiplexed with other operations. But it represents a single, long-running operation for the application itself. This model is not just a straight-forward one for developers. It also results in a time span where the application process issuing the I/O operation does not require CPU time. This is a convenient time for the operating system scheduler to switch to other processes.

synchronous non-blocking I/O In this mode, the application accesses the I/O device in a non-blocking mode. As a result, the kernel space immediately returns the I/O call. Usually, the device is not yet ready and the call response indicates that the call should be repeated later. By doing this, the application code often implements a busy-wait behavior, which can be extremely inefficient. Once the I/O operations has finished and the data is available in user space, the application can continue to run and use the data (in case of a read operation).

asynchronous blocking I/O Surprisingly, the asynchronous blocking model does still use a non-blocking mode for I/O operations. However, instead of a busy-wait, a dedicated blocking system call is used for notifications of the I/O state. Several system calls provide such a functionality, including select, poll, epoll and kqueue [Ste03]. In so doing, multiple I/O descriptors can be passed to the system call. When the implementation of the blocking system call for notifications is sound and performant, this is good model for highly concurrent I/O.

asynchronous non-blocking I/O This I/O model immediately returns from I/O calls. On completition, an event is emitted or a callback is executed. The interesting characteristics of this model is the fact there is no blocking or busy-waiting on user level. The entire operation is shifted to the kernel space. This allows the application to take advantage of additional CPU time while the I/O operations happens in the background on kernel level. In other words, the application can overlap the I/O operations with additional CPU-bound operations or dispatch additional I/O operations in the meantime. Unsurprisingly, this model also provides good performance under highly concurrent I/O.

	blocking	non-blocking
synchronous	`read/write`	`read/write` using `O_NONBLOCK`
asynchronous	I/O multiplexing	AIO

Table 4.1: I/O Models in Linux

These models only describe I/O operations for Linux-based operating systems on a low level. From a more abstract programmer's perspective, models can be substituted by others, often with some performance penalties. An application framework can provide I/O access using synchronous blocking via background threads, but provide an asynchronous interface for the developers using callbacks, and vice versa.

From now on, we differentiate the synchronous blocking approach and the other three approaches in most cases. Being based on some kind of signaling, notification or callback execution, we refer to the latter as event-driven or event-based approaches.

4.2 Server Architectures

We have seen different models for socket I/O—and file I/O, in case of a web server for static content. Now, we are now in need of models merging I/O operations, CPU-bound activities such as request parsing and request handling into general server architectures.

There are traditionally two competitive server architectures—one is based on threads, the other on events. Over time, more sophisticated variants emerged, sometimes combining both approaches. There has been a long controversy, whether threads or events are generally the better fundament for high performance web servers [Ous96, vB03a, Wel01]. After more than a decade, this argument has been now reinforced, thanks to new scalability challenges and the trend towards multi-core CPUs.

Before we evaluate the different approaches, we introduce the general architectures, the corresponding patterns in use and give some real world examples.

4.2.1 Thread-based Server Architectures

The thread-based approach basically associates each incoming connection with a separate thread (resp. process). In this way, synchronous blocking I/O is the natural way of dealing with I/O. It is a common approach that is well supported by many programming languages. It also leads to a straight forward programming model, because all tasks necessary for request handling can be coded sequentially. Moreover, it provides a simple mental abstraction by isolating requests and hiding concurrency. Real concurrency is achieved by employing multiple threads/processes at the same time.

Conceptually, multi-process and multi-threaded architectures share the same principles: each new connection is handled by a dedicated activity.

Multi-Process Architectures

A traditional approach to UNIX-based network servers is the process-per-connection model, using a dedicated process for handling a connection [Ste03]. This model has also been used for the first HTTP server, CERN httpd[1]. Due to the nature of processes, they are isolating different requests promptly, as they do not share memory. Being rather heavyweight structures, the creation of processes is a costly operation and servers often employ a strategy called preforking. When using preforking, the main server process forks several handler processes preemptively on start-up, as shown in figure 4.1. Often, the (thread-safe) socket descriptor is shared among all processes, and each process blocks for a new connection, handles the connection and then waits for the next connection.

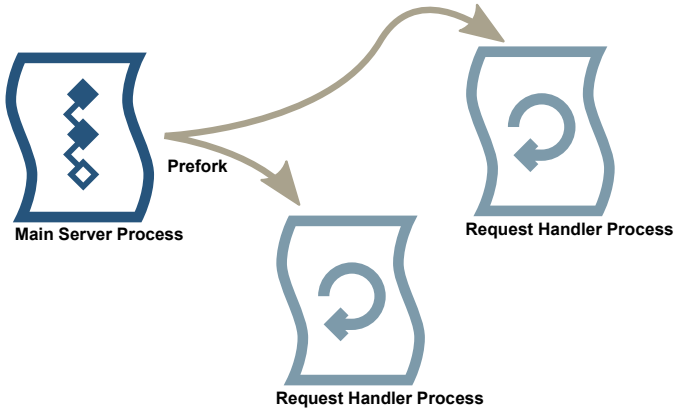

Figure 4.1: A multi-process architecture that make use of preforking. On startup, the main server process forks several child processes that will later handle requests. A socket is created and shared between the processes. Each request handler process waits for new connections to handle and thereafter blocks for new connections.

Some multi-process servers also measure the load and spawn additional requests when needed. However, it is important to note that the heavyweight structure of a process limits the maximum of simultaneous connections. The large memory footprint as a result of the connection-process mapping leads to a concurrency/memory trade-off. Especially in case of long-running, partially inactive connections (e.g. long-polling notification requests), the multi-process architecture provides only limited scalability for concurrent requests.

The popular Apache web server provides a robust multi-processing module that is based on process preforking, Apache-MPM prefork. It is still the default multi-processing module for UNIX-based setups of Apache.

1 `http://www.w3.org/Daemon/Implementation/`

Multi-Threaded Architectures

When reasonable threading libraries have become available, new server architectures emerged that replaced heavyweight processes with more leightweight threads. In effect, they employ a thread-per-connection model. Although following the same principles, the multi-threaded approach has several important differences. First of all, multiple threads share the same address space and hence share global variables and state. This makes it possible to implement mutual features for all request handlers, such as a shared cache for cacheable responses inside the web server. Obviously, correct synchronization and coordination is then required. Another difference of the more leightweight structures of threads is their smaller memory footprint. Compared to the full-blown memory size of an entire process, a thread only consumes limited memory (i.e. the thread stack). Also, threads require less resources for creation/termination. We have already seen that the dimensions of a process are a severe problem in case of high concurrency. Threads are generally a more efficient replacement when mapping connections to activities.

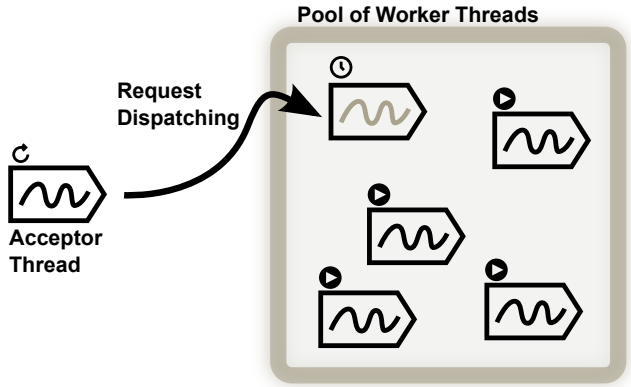

Figure 4.2: A multi-threaded architecture that make use of an acceptor thread. The dedicated acceptor blocks for new socket connections, accepts connections and dispatches them to the worker pool and continues. The worker pool provides a set of threads that handle incoming requests. Worker threads are either handling requests or waiting for new requests to process.

In practice, it is a common architecture to place a single dispatcher thread (sometimes also called acceptor thread) in front of a pool of threads for connection handling [Ste03], as shown in figure 4.2. Thread pools are a common way of bounding the maximum number of threads inside the server. The dispatcher blocks on the socket for new connections. Once established, the connection is passed to a queue of incoming connections. Threads from the thread pool take connections from the queue, execute the requests and wait for new connections in the queue. When the queue is also bounded, the maximum number of awaiting connections can be restricted. Additional connections will be rejected. While this strategy limits the concurrency, it provides more predictable latencies and prevents total overload.

Apache-MPM worker is a multi-processing module for the Apache web server that combines processes and threads. The module spawns several processes and each process in turn manages its own pool of threads.

Scalability Considerations for Multi-Threaded Architectures

Multi-threaded servers using a thread-per-connection model are easy to implement and follow a simple strategy. Synchronous, blocking I/O operations can be used as a natural way of expressing I/O access. The operating system overlaps multiple threads via preemptively scheduling. In most cases, at least a blocking I/O operation triggers scheduling and causes a context switch, allowing the next thread to continue. This is a sound model for decent concurrency, and also appropriate when a reasonable amount of CPU-bound operations must be executed. Furthermore, multiple CPU cores can be used directly, as threads and processes are scheduled to all cores available.

Under heavy load, a multi-threaded web server consumes large amounts of memory (due to a single thread stack for each connection), and constant context switching causes considerable losses of CPU time. An indirect penalty thereof is increased chance of CPU cache misses. Reducing the absolute number of threads improves the per-thread performance, but limits the overall scalability in terms of maximum simultaneous connections.

4.2.2 Event-driven Server Architectures

As an alternative to synchronous blocking I/O, the event-driven approach is also common in server architectures. Due to the asynchronous/non-blocking call semantics, other models than the previously outlined thread-per-connection model are needed. A common model is the mapping of a single thread to multiple connections. The thread then handles all occurring events from I/O operations of these connections and requests. As shown in figure 4.3, new events are queued and the thread executes a so-called event loop—dequeuing events from the queue, processing the event, then taking the next event or waiting for new events to be pushed. Thus, the work executed by a thread is very similar to that of a scheduler, multiplexing multiple connections to a single flow of execution.

Processing an event either requires registered event handler code for specific events, or it is based on the execution of a callback associated to the event in advance. The different states of the connections handled by a thread are organized in appropriate data structures— either explicitly using finite state machines or implicitly via continuations or closures of callbacks. As a result, the control flow of an application following the event-driven style is somehow inverted. Instead of sequential operations, an event-driven program uses a cascade of asynchronous calls and callbacks that get executed on events. This notion often makes the flow of control less obvious and complicates debugging.

The usage of event-driven server architectures has historically depended on the availability of asynchronous/non-blocking I/O operations on OS level and suitable, high performance event

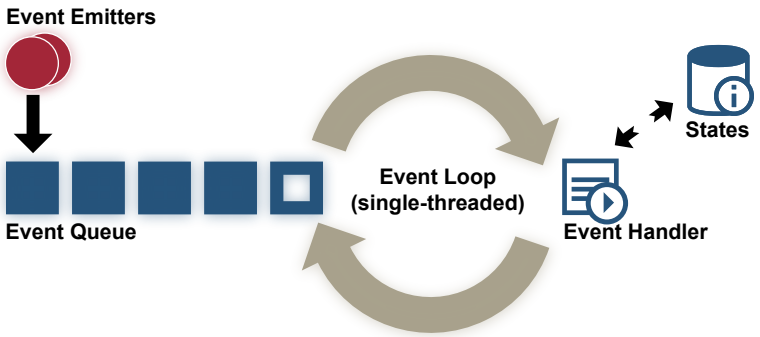

Figure 4.3: This conceptual model shows the internals of an event-driven architecture. A single-threaded event loop consumes event after event from the queue and sequentially executes associated event handler code. New events are emitted by external sources such as socket or file I/O notifications. Event handlers trigger I/O actions that eventually result in new events later.

notification interfaces such as epoll and kqueue. Earlier implementations of event-based servers such as the Flash web server by Pai et al [Pai99].

Non-blocking I/O Multiplexing Patterns

Different patterns have emerged for event-based I/O multiplexing, recommending solutions for highly concurrent, high-performance I/O handling. The patterns generally address the problem of network services to handle multiple concurrent requests.

Reactor Pattern The Reactor pattern [Sch95] targets *synchronous, non-blocking I/O* handling and relies on an event notification interface. On startup, an application following this pattern registers a set of resources (e.g. a socket) and events (e.g. a new connection) it is interested in. For each resource event the application is interested in, an appropriate event handler must be provided—a callback or hook method. The core component of the Reactor pattern is a synchronous event demultiplexer, that awaits events of resources using a blocking event notification interface. Whenever the synchronous event demultiplexer receives an event (e.g. a new client connection), it notifies a dispatcher and awaits for the next event. The dispatcher processes the event by selecting the associated event handler and triggering the callback/hook execution.

The Reactor pattern thus decouples a general framework for event handling and multiplexing from the application-specific event handlers. The original pattern focuses on a single-threaded execution. This requires the event handlers to adhere to the non-blocking style of operations. Otherwise, a blocking operation can suspend the entire application. Other variants of the Reactor

pattern use a thread pool for the event handlers. While this improves performance on multi-core platforms, an additional overhead for coordination and synchronization must be taken into account.

Proactor Pattern In contrast, the Proactor pattern [Pya97] leverages truly *asynchronous, non-blocking I/O* operations, as provided by interfaces such as POSIX AIO[1]. As a result, the Proactor can be considered as an entirely asynchronous variant of the Reactor pattern seen before. It incorporates support for completition events instead of blocking event notification interfaces. A proactive initiator represents the main application thread and is responsible for initiating asynchronous I/O operations. When issuing such an operation, it always registers a completition handler and completition dispatcher. The execution of the asynchronous operation is governed by the asynchronous operation processor, an entity that is part of the OS in practice. When the I/O operation has been completed, the completition dispatcher is notified. Next, the completition handler processes the resulting event.

An important property in terms of scalability compared to the Reactor pattern is the better multithreading support. The execution of completition handlers can easily be handed off to a dedicated thread pool.

Scalability Considerations for Event-driven Architectures

Having a single thread running an event loop and waiting for I/O notifications has a different impact on scalability than the thread-based approach outlined before. Not associating connections and threads does dramatically decrease the number of threads of the server—in an extreme case, down to the single event-looping thread plus some OS kernel threads for I/O. We thereby get rid of the overhead of excessive context switching and do not need a thread stack for each connection. This decreases the memory footprint under load and wastes less CPU time to context switching. Ideally, the CPU becomes the only apparent bottleneck of an event-driven network application. Until full saturation of resources is archived, the event loop scales with increasing throughput. Once the load increases beyond maximum saturation, the event queue begins to stack up as the event-processing thread is not able to match up. Under this condition, the event-driven approach still provides a thorough throughput, but latencies of requests increase linearly, due to overload. This might be acceptable for temporary load peaks, but permanent overload degrades performance and renders the service unusable. One countermeasure is a more resource-aware scheduling and decoupling of event processing, as we will see soon when analysing a staged-based approach.

For the moment, we stay with the event-driven architectures and align them with multi-core architectures. While the thread-based model covers both—I/O-based and CPU-based

1 `http://www.kernel.org/doc/man-pages/online/pages/man7/aio.7.html`

concurrency, the initial event-based architecture solely addresses I/O concurrency. For exploiting multiple CPUs or cores, event-driven servers must be further adapted.

An obvious approach is the instantiation of multiple separate server processes on a single machine. This is often referred to as the *N*-copy approach for using *N* instances on a host with *N* CPUs/cores. In our case a machine would run multiple web server instances and register all instances at the load balancers. A less isolated alternative shares the server socket between all instances, thus requiring some coordination. For instance, an implementation of this approach is available for node.js using the cluster module[1], which forks multiple instances of an application and shares a single server socket.

The web servers in the architectural model have a specific feature—they are stateless, shared-nothing components. Already using an internal cache for dynamic requests requires several changes in the server architecture. For the moment, the easier concurrency model of having a single-threaded server and sequential execution semantics of callbacks can be accepted as part of the architecture. It is exactly this simple execution model that makes single-threaded applications attractive for developers, as the efforts of coordination and synchronization are diminished and the application code (i.e. callbacks) is guaranteed not to run concurrently. On the other hand, this characteristic intrinsically prevents the utilization of multiple processes inside a single event-driven application. Zeldovich et al. have addresses this issue with *libasync-smp* [Zel03], an asynchronous programming library taking advantage of multiple processes and parallel callback execution. The simple sequential programming model is still preserved. The basic idea is the usage of tokens, so-called colors assigned to each callback. Callbacks with different colors can be executed in parallel, while serial execution is guaranteed for callbacks with the same color. Using a default color to all non-labelled callbacks makes this approach backward compatible to programs without any colors.

Let us extend our web server with a cache, using the coloring for additional concurrency. Reading and parsing a new request are sequential operations, but different requests can be handled at the same time. Thus, each request gets a distinct color (e.g. using the socket descriptor), and the parsing operation of different request can actually happen in parallel, as they are labelled differently. After having parsed the request, the server must check if the required content is already cached. Otherwise, it must be requested from the application server. Checking the cache now is a concurrent operation that must be executed sequentially, in order to provide consistency. Hence, the same color label is used for this step for all requests, indicating the scheduler to run all of these operations always serially, and never in parallel. This library also allows the callback to execute partially blocking operations. As long as the operation is not labelled with a shared color, it will not block other callbacks directly. The library is backed by a thread pool and a set of event queues, distinguished by colors. This solution allows to adhere to the traditional event-driven programming style, but introduces real concurrency to a certain extent. However, it requires the

1 `http://nodejs.org/docs/latest/api/cluster.html`

developer to label callbacks correctly. Reasoning about the flows of executions in an event-driven program is already difficult sometimes, and the additional effort may complicate this further.

4.2.3 Combined Approaches

The need for scalable architectures and the drawbacks of both general models have led to alternative architectures and libraries incorporating features of both models.

Staged Event-driven Architecture

A formative architecture combining threads and events for scalable servers has been designed by Welsh et al. [Wel01], the so called Staged Event-driven Architecture (SEDA). As a basic concept, it divides the server logic into a series of well-defined stages, that are connected by queues, as shown in figure 4.4. Requests are passed from stage to stage during processing. Each stage is backed by a thread or a thread pool, that may be configured dynamically.

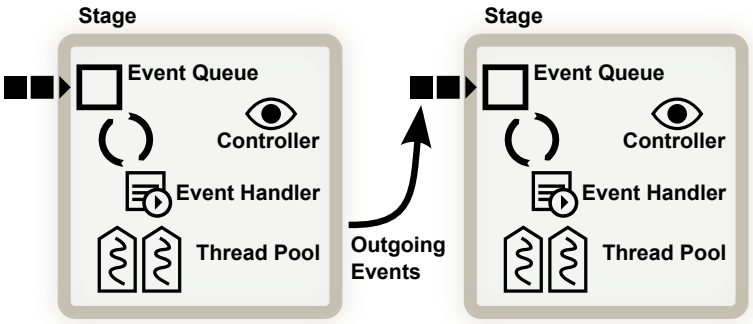

Figure 4.4: This illustration shows the concept of SEDA. In this example, there are two stages, each with a queue for incoming events, an event handler backed by thread pool and a controller that monitors resources. The only interaction between stages is the emission of events to the next stage(s) in the pipeline.

The separation favors modularity as the pipeline of stages can be changed and extended easily. Another very important feature of the SEDA design is the resource awareness and explicit control of load. The size of the enqueued items per stage and the workload of the thread pool per stage gives explicit insights on the overall load factor. In case of an overload situation, a server can adjust scheduling parameters or thread pool sizes. Other adaptive strategies include dynamic reconfiguration of the pipeline or deliberate request termination. When resource management, load introspection and adaptivity are decoupled from the application logic of a stage, it is simple to develop well-conditioned services. From a concurrency perspective, SEDA represents a hybrid

approach between thread-per-connection multithreading and event-based concurrency. Having a thread (or a thread pool) dequeuing and processing elements resembles an event-driven approach. The usage of multiple stages with independent threads effectively utilizies multiple CPUs or cores and tends to a multi-threaded environment. From a developer's perspective, the implementation of handler code for a certain stage also resembles more traditional thread programming.

The drawbacks of SEDA are the increased latencies due to queue and stage traversal even in case of minimal load. In a later retrospective [Wel10], Welsh also criticized a missing differentiation of module boundaries (stages) and concurrency boundaries (queues and threads). This distribution triggers too many context switches, when a requests passes through multiple stages and queues. A better solution groups multiple stages together with a common thread pool. This decreaes context switches and improves response times. Stages with I/O operations and comparatively long execution times can still be isolated.

The SEDA model has inspired several implementations, including the generic server framework Apache MINA[1] and enterprise service buses such as Mule ESB[2].

Special-Purpose Libraries

Other approaches focused on the drawbacks of threads in general and the problems of available (user-level) threading libraries in particular. As we will see soon, most of the scalability problems with threads are associated with shortcomings of their libraries.

For instance, the Capriccio threading library by von Behren et al. [vB03b] promises scalable threads for servers by tackling the main thread issues. The problem of extensive context switches is addressed by using a non-preemptive scheduling. Threads eithers yield on I/O operations, or on an explicit yield operation. The stack size of each thread is limited based on prior analysis at compile time. This makes it unnecessary to overprovide bounded stack space preemptively. However, unbounded loops and the usage of recursive calls render a complete calculation of stack size apriori impossible. As a workaround, checkpoints are inserted into the code, that determine if a stack overflow is about to happen and allocate new stack chunks in that case. The checkpoints are inserted at compile time and are placed in a manner that there will never be a stack overflow within the code between two checkpoints. Additionally, resource-aware scheduling is applied that prevents thrashing. Therefore, CPU, memory and file descriptors are watched and combined with a static analysis of the resource usage of threads, scheduling is dynamically adapted.

Also, hybrid libraries, combining threads and events, have been developed. Li and Zdancewic [Li07] have implemented a combined model for Haskell, based on concurrency monads. The programming language Scala also provides event-driven and multi-threaded concurrency, that can be combined for server implementations.

1 http://http://mina.apache.org
2 http://www.mulesoft.com/

	thread-based	event-driven
connection/request state	thread context	state machine/continuation
main I/O model	synchronous/blocking	asynchronous/non-blocking
activity flow	thread-per-connection	events and associated handlers
primary scheduling strategy	preemptive (OS)	cooperative
scheduling component	scheduler (OS)	event loop
calling semantics	blocking	dispatching/awaiting events

Table 4.2: Main differences between thread-based and event-driven server architectures.

4.2.4 Evaluation

So far, we have regarded different architectural principles for building concurrent web servers. When implementing a server for highly concurrent usage, one of these models should be applied. However, there are other factors that also influence the actual performance and scalability of the implementation. This includes the programming language, the execution environment (e.g. virtual machine) the operating system, the thread libraries that can be used and the available means for I/O operations (e.g. support for true asynchronous I/O). For each server architecture, scalable server implementations can be implemented—however, the actual requirements differ.

Pariag et al. [Par07] have conducted a detailed performance-oriented comparison of thread-based, event-driven and hybrid pipelined servers. The thread-based server (knot) has taken advantage of the aforementioned Capriccio library. The event-driven server (μserver) has been designed to support socket sharing and multiprocessor support using the N-copy approach. Lastly, the hybrid pipelined server (WatPipe) has been heavily inspired by SEDA, and consists of four stages for serving web requests. Pariag and his team then tested and heavily tuned the three servers. Finally, they benchmarked the servers using different scenarios, including deliberate overload situations. Previous benchmarks have been used to promote either new thread-based or event-driven architectures[Pai99, Wel01, vB03a], often with clear benefits for the new architecture. The extensive benchmark of Pariag et al. revealed that all three architectural models can be used for building highly scalable servers, as long as thorough tuning and (re-)configuration is conducted. The results also showed that event-driven architectures using asynchronous I/O have still a marginal advantage over thread-based architectures.

Event-driven web servers like nginx (e.g. GitHub, WordPress.com), lighttpd (e.g. YouTube, Wikipedia) or Tornado[1] (e.g. Facebook, Quora) are currently very popular and several generic frameworks have emerged that follow this architectural pattern. Such frameworks available for Java include netty[2] and MINA.

1 http://www.tornadoweb.org/
2 http://www.jboss.org/netty

Please note that we do not conduct our own benchmarks in this chapter. Nottingham, one of the editors of the HTTP standards, has written an insightful summary, why even handed server benchmarking is extremely hard and costly [Not11]. Hence, we solely focus on the architecture concepts and design principles of web servers and confine our considerations to the prior results of Pariag et al. [Par07].

4.3 The Case of Threads vs. Events

We have outlined a special set of requirements for our servers such as statelessness. This eases not just actual implementations, but it also biases our considerations of threads vs. events to some extent. In chapter 5, we will focus on concurrent programming from a more general perspective. For the rest of this chapter, we take some time for a closer look on the general argument of threads vs. events. The discussion is a very old one that emerged long ago in the field of operating systems. However, high performance network servers have always been a challenging topic and the argument has been repeatedly revisited several times in this context.

We introduce the duality argument of Lauer and Needham that reasons the intrinsic relationship between threads and events. Then, we review some of the more recent publications comparing both models or campaigning for one of them, often painting black the other one at the same time. Finally, we provide a neutral view on both models and conclude on their strengths and weaknesses.

4.3.1 The Duality Argument

In the late seventies, Lauer and Needham [Lau79] took an extensive look on two predominant programming models for operating system designs in terms of processes and synchronization and communication. More precisely, they compare message-oriented systems with procedure-oriented systems. The former uses a small number of processes that use explicit messaging, and the latter is based on large numbers of small processes using shared data instead. Thus, message-oriented systems resemble event-driven systems, while procedure-oriented systems correspond to thread-based systems [vB03a, Li07]. Their main contribution are three important observations:

1. Both models are *duals of each other*. A program written in one model can be mapped directly to an equivalent program based on the other model.

2. Both models are *logically equivalent*, although they use diverging concepts and provide a different syntax.

3. The *performance* of programs written in both models is essentially *identical*, given that identical scheduling strategies are used.

Consequently, Lauer and Needham come up with a mapping of both models, allowing to confront building blocks of both concepts. The most important mappings are shown in table 4.3.

The flow of control in an application using either one of the models generates a unique graph that contains certain yielding or blocking nodes (e.g. awaiting a reply resp. a procedure return). The edges between such nodes represent code that is executed when traversing the graph. According to the duality argument, both thread-based and event-driven programs yield the same blocking points, when equivalent logic is implemented and the programs are thus duals. Von Behren refers to this graph representation as a blocking graph [vB03a].

Lauer and Needham argue that by replacing concepts and transforming a program from one model into the other, the logic is not affected and the semantic content of the code is thus invariant. As a result, they claim that both conceptual models are equivalent and even the performance of both models is the same, given a proper execution environment. As a consequence, they suggest that the choice of the right model depends on the actual application, and neither model is preferable in general.

While the general concepts are accepted to be comparable [vB03a, Li07], there is also some criticism to the mapping, especially when it is applied to event-based systems that mix in other programming concepts. Von Behren [vB03a] points out that Lauer and Needham ignore cooperative scheduling for event-based systems, which is an important part of many event-driven systems today. Lauer and Needham also disallow any kind of shared memory or global data in their mapping. But many event-driven systems indeed use shared memory in a few places.

Despite these remarks, the duality argument generally allows us to relax our considerations on both systems in terms of intrinsic performance characteristics. In fact, we can focus on the applicability and suitability of both models based on actual requirements of an application. Furthermore, the duality argument motivates us to question the implementation of the models instead of the models themselves when it comes to performance and scalability.

Next, we have a look at some popular pleadings for one model or the other, complemented by a compilation of prevalent criticism for each model.

thread-based		**event-driven**
monitor	~	event handler
scheduling	~	event loop
exported functions	~	event types accepted by event handler
returning from a procedure	~	dispatching a reply
executing a blocking procedure call	~	dispatching a message, awaiting a reply
waiting on condition variables	~	awaiting messages

Table 4.3: A mapping of thread-based and event-driven concepts based on Lauer and Needham [Lau79], adapted and restated to resemble event-driven systems [vB03a, Li07].

4.3.2 A Case for Threads

Proponents of threads argue that threads are the natural extension of the dominant sequential programming style in most programming languages for providing concurrency [vB03a, vB03b, Gus05]. From a developer perspective, threads map work to be executed with associated flows of control. More precisely, a thread represents work from the perspective of the task itself. This allows to develop concurrent code while focusing on the sequential steps of operations required to complete the task. Transparently executing blocking operations and I/O calls relieves the developer from low-level scheduling details. Instead, he can rely on the operating system and the runtime environment.

Threads are well-known and understood entities of operating systems and are general purpose primitives for any kind of parallelism. Threads are also mandatory for exploiting true CPU concurrency. Hence, even other concurrency approaches rely on underlying, thread-based implementations, although they hide this trait from the developer.

The abstraction that threads provide appears to be simple and especially powerful when tasks are mostly isolated and only share a limited amount of state (cf. multi-threaded web servers). Threads also provide a solid structuring primitive for concurrent applications in terms of syntax.

The opponents of thread-based systems line up several drawbacks. For Ousterhout, who probably published the most well-known rant against threads [Ous96], the extreme difficulty of developing correct concurrent code—even for programming experts—is the most harmful trait of threads. As soon as a multi-threaded system shares a single state between multiple threads, coordination and synchronization becomes an imperative. Coordination and synchronization requires locking primitives, which in turn brings along additional issues. Erroneous locking introduces deadlocks or livelocks, and threatens the liveness of the application. Choosing the right locking granularity is also source of trouble. Too coarse locks slow down concurrent code and lead to degraded sequential execution. By contrast, too fine locks increase the danger of deadlocks/livelocks and increase locking overhead. Concurrent components based on threads and locks are not composable. Given two different components that are thread-safe, a composition of them is not thread-safe per se. For instance, placing circular dependencies between multi-threaded components unknowingly can introduce severe deadlocks.

Lee [Lee06] focuses on the lack of understandability and predictability of multi-threaded code, due to nondeterminism and preemptive scheduling. Multithreading appears to be error-prone, and very difficult to debug. The state explosion as a result from all possible interleavings of multiple threads renders a reasonable execution analysis of concurrent code virtually impossible. This is primarily caused by the unpredictability of preemptive scheduling. So, contrary to von Behren [vB03a], Lee argues that threads are precisely not a good abstraction for concurrent flows of execution. Quite the opposite, the oversimplifying abstraction of threads appears to be misleading, as it pretends a continuous execution of code that may not match any real runtime behavior.

Concerning performance, we have already got to know the downside of extensive context switching. Similarly, huge numbers of threads require large amounts of memory due to their thread stacks. The usage of locks yields an additional overhead.

In return, the pro-thread camp argues that a couple of the drawbacks mentioned are actually the result of poor threading library implementations and the essence of preemptive scheduling.

4.3.3 A Case for Events

The campaigners for events like Ousterhout[Ous96] regard event-driven systems as the more appropriate foundation for high concurrency servers when compared to thread-based systems. The fundamental idea of a single-threaded event loop eases concurrency concerns by providing a simple, and straight model of parallelism.

By not using blocking/synchronous I/O, multiple I/O operations overlap, although a single thread is used. This enables I/O parallelism without requiring CPU parallelism at the same time. This yields the illusion of multi-threaded concurrency, because multiple conceptual flows of execution appear to happen at the same time (at least their I/O operations do). Event handler code and callbacks can be developed without the immanent fear of concurrent access on state. The execution of a callbacks is guaranteed to be deterministic, as long as no yielding operation is triggered in the callback. This provides a feeling of deterministic reasoning. Scheduling becomes an explicit operation and happens inside the application itself. Fine-grained tuning of scheduling is possible and can take into account application-specific requirements.

The usage of events and event handlers yields an asynchronous behavior, which is favored by some developers. Instead of giving the abstraction of an isolated sequential flow of executions, the asynchronous style makes the differences between I/O operations and CPU-bound operations obvious.

However, there are also serious concerns over event-driven systems. The most common reason why event-driven systems are rejected is their programming style. The idea of an event loop and registered event handlers yields an inversion of control. Instead of sequential operations, code is organized as a fragmented set of event handlers and callbacks. In non-trivial applications, this leads to heavy chaining of callbacks. Gustafsson refers to the notion of an event loop sequentially executing callbacks on events as a form of "delayed GOTO" [Gus05]. Compared to threads, that provide higher abstractions, event-driven systems hence appear as a step backwards.

Existing sequential algorithms can not be used directly in event-driven systems. Instead, whenever an I/O operation is triggered, the code must be split up and moved into different callbacks, creating large cascading callback chains.

The obfuscated control flow is often accompanied by the necessity of saving and restoring state before yielding and after resuming. This is especially obvious when imperative, low-level programming languages are used that do not support mitigating language idioms like closures. A thread can store state as part of its thread stack, independently of any scheduling. In an

event-driven system, it is the developer's responsibility to handle and recover state between event handlers.

While threads are endangered by deadlocks or livelocks, single-threaded, event-driven applications can be scuttled by long running, CPU-bound callbacks, blocking operations or callbacks that refuse to yield.

While the single-threaded event-loop model fits for mostly I/O-bound applications, it is very difficult by default to take advantage of real CPU concurrency and utilize multiple cores (cf. subsection 4.2.2).

4.3.4 A Conflation of Distinct Concepts

Another substantial argument for the case of threads vs. events has been made by Adya et al. [Ady02]. Debating about thread-based and event-based programming styles, they derive different management concepts that these programming styles make use of for concurrency. However, they argue that these concepts are often conflated and also confused with the actual programming styles themselves. Adya et al. state that this makes it harder to reason about appropriate approaches towards concurrent programming. The separation of concepts yields five distinct concepts, most of them orthogonal to each other.

Task Management

The flows of execution within a program are often divided into separate tasks that coexist. Managing the concurrent execution of these tasks requires a management concept on how to switch between tasks like scheduling does. Serial task management sequentially runs a task to completion, then switching to the next task. While this strategy prevents state conflicts due to isolated execution, it does not allow to exploit true parallelism. Also, long-running tasks or tasks waiting for I/O will delay the execution of other pending tasks. Preemptive task management instead enables an overlapping execution of multiple tasks at the same time and makes use of multiple cores. However, tasks will be scheduled externally, thus a task is not aware of task management.

An interesting alternative is cooperative task management, preserving some of the advantages of both models. Tasks yield cooperatively and explicitly, but make it still easier to reason about the code. Single-threaded cooperative task management facilitates to deal with invariants and state. For multi-threaded code, cooperative task management often decreases the number of context switches.

Stack Management

Another concept governs the relationship between flows of execution and associated states. In a thread-based model, tasks have their own stacks, hence (automatic) stack management is an inherent feature. Systems based on events require a different handling of task stacks. As the flow of execution of a logical task is in this case represented by a sequence of dispatched events

and corresponding executions of event handlers, there is no direct notion of a stack. Moreover, different procedures handle the sequence of events corresponding to the logical task, so state handling must be broken across several event handlers. As a result, the stack must be provided explicitly by the developer. Adya et al. refer to this duty as *stack ripping*. Before yielding, the stack of a task must be serialized and stored. When an event handler later continues the execution, it must first load and reconstruct the stack of the corresponding task.

Some functional languages such as Scheme provide languages idioms for that, like closures or continuations. Closures are functions that encapsulate their referencing environment (i.e. "stack"). Continuations are special closures used for encapsulating control state. Most low-level languages such as C do not support these functional mechanisms, and stack ripping therefore remains as a mandatory workaround.

I/O Management

I/O management is responsible for the I/O operations, and can be separated into synchronous and asynchronous management interfaces. We have already considered both concepts in detail earlier in this chapter. However, it is important to notice that I/O management and task management are orthogonal concepts. While computational operations may share state between tasks, this is generally not true for I/O. Consequently, tasks executing I/O operations concurrently can be overlapped. Furthermore, each of the task management concepts can be used either with a synchronous or asynchronous I/O management concept.

Conflict Management

Different task management concepts provide specific agreements on the granularity of atomicity of operations. This is important for guaranteeing consistent data when state is shared between tasks. Serial and to some extent (i.e. single-threaded) cooperative task management concepts provide a very simple form of conflict management. Serial tasks are exclusively executed, and cooperative tasks provide atomic semantics between all yielding operations. This makes it very easy to reason about invariants. Ensuring that invariants hold is more complex for preemptive task management and requires synchronization mechanisms.

Data Partitioning

We have seen that shared state and the preservation of consistency correlates to both task and conflict management. As a result, partitioning data and restrictively allowing access to state may reduce the possibilities of conflicts. For instance, thread-local state does not have to be shared and can be partitioned explicitly.

Looking for the Sweet Spots

Adya et al. propose the separation of management concepts to argue purposefully for the most convenient form of concurrent programming, aside from the coarse thread vs. event debate [Ady02]. They pay special attention to the first two management principles. While traditional event-based systems are mostly based on cooperative task management and manual stack management requiring stack ripping, thread-based systems often use preemptive task management and automatic stack management. Eventually, they favor a model that makes use of cooperative task management, but releases the developer from the burden of stack management, as shown in figure 4.5. Such a model eases concurrency reflections, requires minimal conflict management and harmonizes with both I/O management models.

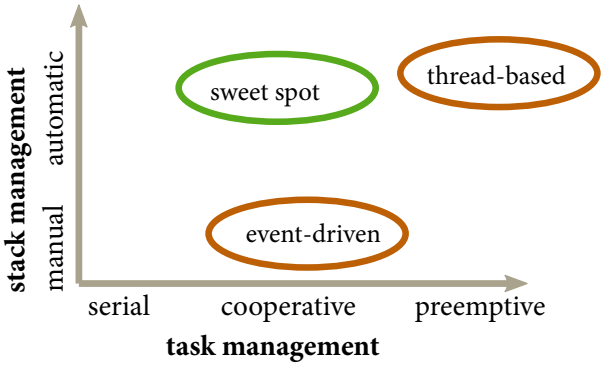

Figure 4.5: The sweet spot of task management and stack management by using cooperative task management and automatic stack management, according to Adya et al. [Ady02].

Some of the recent event-driven systems such as node.js come very close to this intended model. They rely on closures as language primitives that encapsulate stack data into callback functions, mitigating stack ripping efforts.

Gustafsson [Gus05] comes to a similar result. It is not the nature of threads that makes their usage cumbersome, but preemptive scheduling. A cooperative scheduling of threads eases much of the pain of threads. Non-preemptive scheduling let us preserve invariants without extensive locking. Gustafsson also backs Adya by noting that the question of threads or events is orthogonal to the question of cooperative or preemptive scheduling.

4.3.5 Conclusion

Before we value the different models, let us restate our original question. We are in search of appropriate programming models for high concurrency and high performance network servers,

in our case web servers.

We had a detailed look on both thread-based and event driven approaches. We also learned about hybrid approaches such as SEDA, and highly optimized threading libraries that employ compiler optimizations for thread stack sizes and adaptive scheduling. We have seen that various servers using different approaches can be tuned and optimized to give at least roughly a similar performance.

However, for large-scale connection concurrency, event-driven server architectures using asynchronous/non-blocking I/O operations seem to be more popular, as they provide a slightly better scalability under heavy concurrency. Such servers demand less memory, even when they handle thousands of concurrent connections. Also, they do not require specialized threading libraries. On the other hand, mature threading implementations such as the Native POSIX Thread Library [Molo3] still provide reasonable performance, even for highly concurrent server implementations.

Concerning the everlasting argument between the pro-threads camp and pro-events camp, we have seen that both programming models are actually duals of each other and can be transformed under certain restrictions. So the actual performances of a server using either one of the models depends to a large extent on the real environment it is executed on, including the operating system and hardware features. As the duality argument dates back to a time where asynchronous, non-blocking I/O operations, multi-core CPUs have not been available yet, the influence of the environment must not be underestimated.

Next, we took a look behind the essence of threads and events, realizing that the different management concepts are often conflated, when arguing about both models. It is primarily the cooperative scheduling nature that makes event-driven systems so interesting for highly concurrent servers. The downside of many low-level event-driven systems is the required effort for stack ripping, a concept that is not necessary for threads, as the thread stack frame encapsulates state. Today, functional and multi-paradigm languages mitigate the problem of stack ripping by language idioms like closures. This allows a more decent programming style in event-driven systems, although the style is still very different compared to the sequential structuring of threads.

4.4 Summary

The challenge of scalability for web servers is characterized by intense concurrency of HTTP connections. The massive parallelism of I/O-bound operations is thus the primary issue. When multiple clients connect to a server simultaneously, server resources such as CPU time, memory and socket capacities must be strictly scheduled and utilized in order to maintain low response latencies and high throughput at the same time. We have therefore examined different models for I/O operations and how to represent requests in a programming model supporting concurrency. We have focused on various server architectures that provide different combinations of the aforementioned concepts, namely multi-process servers, multi-threaded servers, event-driven

servers and combined approaches such as SEDA.

The development of high performance servers using either threads, events, or both emerged as a viable possibility. However, the traditional synchronous, blocking I/O model suffers a performance setback when it is used as part of massive I/O parallelism. Similarly, the usage of large numbers of threads is limited by increasing performance penalties as a result of permanent context switching and memory consumption due to thread stack sizes. On the other hand, event-driven server architectures suffer from a less comprehensible and understandable programming style and can often not take direct advantage of true CPU parallelism. Combined approaches attempt to specifically circumvent inherent problems of one of the models, or they suggest concepts that incorporate both models.

We have now seen that thread-based and event-driven approaches are essentially duals of each other and have been dividing the network server community for a long time. Gaining the benefits of cooperative scheduling and asynchronous/non-blocking I/O operations is among the main desires for I/O-bound server applications—however this is often overlooked in a broader and conflated argument between the thread camp and the event camp.

5 Concurrency Concepts for Applications and Business Logic

The previous chapter 4 has dealt with connection concurrency as an issue for web servers. The challenge is characterized by massively I/O-bound operations, but very limited mutable state. In this chapter, we have a look at concurrency from a different angle, by focusing on application servers. That is, the component responsible for executing the actual business logic of an application, stimulated by incoming requests.

The inherent parallelism of requests in a large-scale web architecture is inevitable. Multiple users access the application at the same time, creating large numbers of independent requests. In consequence, application servers are components that must cope with a high degree of concurrency. The main issues we want to address in this chapter are the implications and consequences of different concurrency paradigms when used for business logic of application servers. This not just includes the impact of handling state in concurrent applications, but also the simplicity and accessibility of the particular paradigms for developers.

Moreover, we are not focusing on specific application server implementations or web application frameworks. Instead, we have a more general showdown with concurrency and paradigms for concurrent programming. The reflections are applicable for distributed and concurrent programming in general. Eventually, we are in search of an appropriate programming abstraction for the inherent concurrency of an application that allows us to develop scalable and performant applications, but tames the trouble of concurrency at the same time.

5.1 Overview

An application server provides a reactive behavior that generates responses as a result to incoming requests. The application server component receives invocations from upstream web servers in form of high-level messages (alternatively RPC calls or similar mechanisms). It is thus decoupled from low-level connection handling or request parsing duties. In order to generate a proper response, the application server executes business logic mapped to request URIs. The flow of execution for a request inside an application server includes interactions with different

components of the architecture, such as databases and backend services, but also computational operations as part of the application logic:

CPU-bound activities CPU-bound activities are tasks that primarily consume CPU time during execution. In general, these tasks are computationally heavy algorithms operating on in-memory data. In terms of a web applications, this applies to tasks such as input validation, template rendering or on-the-fly encoding/decoding of content.

I/O-bound activities I/O-bound activities are tasks mainly limited by I/O resources, such as network I/O or file I/O. I/O-bound activities often take place when tasks operates on external data that is not (yet) part of its own memory. In case of our architectural model, this includes access to most platform components, including storage backends, background services and external services.

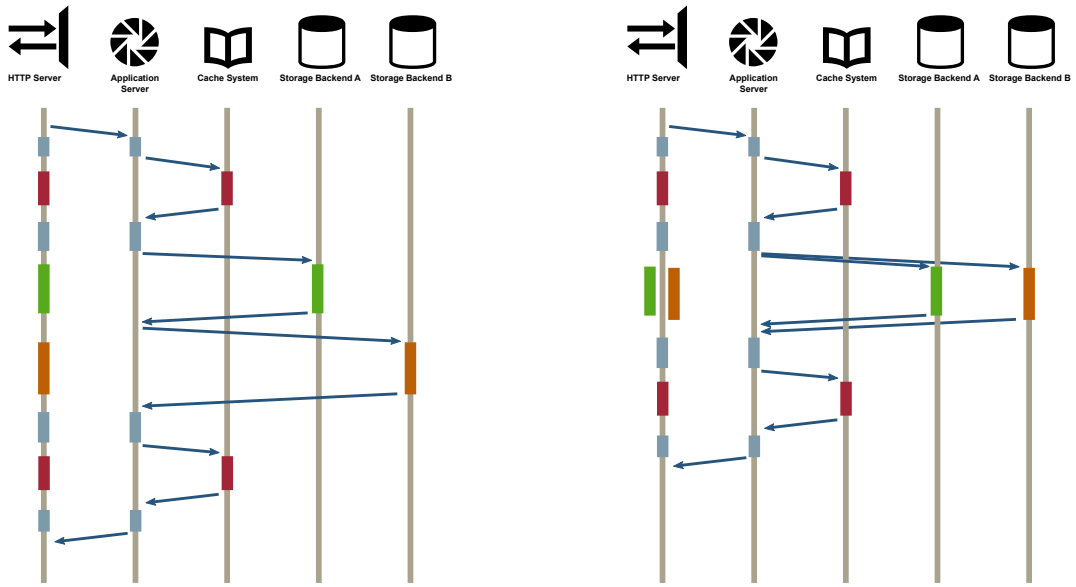

Figure 5.1: A flow of execution for request handling. The application server first queries a cache, then dispatched two independent database queries, and finally accesses the cache again. On the left side, execution is strictly sequential. On the right side, the independent operations are parallelized in order to improve latency results.

A request to a web application typically triggers operations of both types, although the actual ratio depends on the application internals. Many content-rich web applications (e.g. blogs, wikis) primarily rely on database operations, and to some extend on template rendering. Other applications are mainly CPU-bound, like web services that provide computational services or web services for media file conversions. Low latencies of responses are favored in both cases, as they are important for a good user experience.

The application logic associated to a request is often implemented in a sequential order. In order to minimize latencies, the parallelization of independent operations should be considered, as shown in figure 5.1. In a directed graph representation, a first node represents the arrival of the request and a last node the compiled response. Nodes in between represent operations such as database operations or computation functions. Splitting up the flow of control results in parallel operations, that must be synchronized at a later time. This pattern, also known as *scatter and gather* [Hoh03], is particularly viable for I/O-bound activities, such as database operations and access to other architecture components (network I/O).

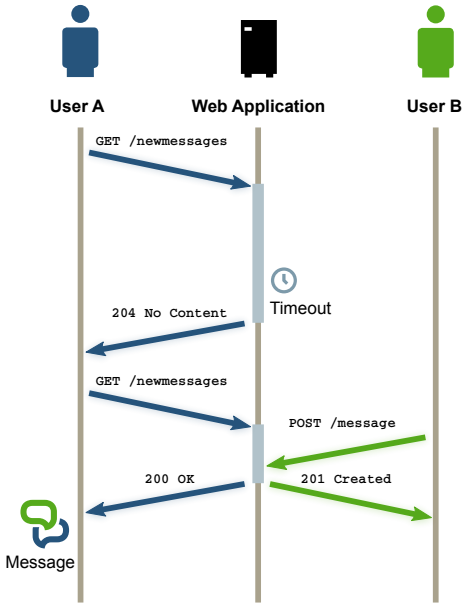

Figure 5.2: A coordination between pending requests as part of an interactive web application. The browser of user A continuously sends requests for notifications by using long polling. Once user B posts a new message, the web application coordinates the notification and responds to both pending requests, effectively notifying user A.

Furthermore, request processing might include coordination and communication with other requests, either using external messaging components, or using a built-in mechanism of the application server component. For instance, this is necessary for collaborative, interactive and web real-time applications. Dedicated requests are then dispatched enabling the server to eventually send a response at a later time triggered by the actions of other requests or other server-side events.

In essence, modern web applications make use of concurrency properties that differ from the notion of entirely isolated requests of earlier applications, as shown in figure 5.2. Application state is not entirely harbored inside database systems anymore, but also shared in other components to some extent, and maybe even between requests. Features for collaboration and interactivity as well

as the demand for low latencies do not allow to eschew synchronization and coordination anymore. Instead, true concurrency must be embraced for the application logic of web applications. For the rest of this chapter, we study different approaches towards concurrent programming and how they manage state, coordination and synchronization. Then, we examine how concurrent web applications can take advantage of these concepts.

5.2 Concurrency Based on Threads, Locks and Shared State

Imperative programming, the most popular form of structured programming, is built around the notion of sequential execution and mutable state. It directly deduces from the conceptions of the Von Neumann architecture. Threads are often regarded as a consequential extension of this notion that enable multiple flows of control simultaneously. Next to heavier processes, threads are the main constructs for parallelism provided by operating systems and hardware architectures (i.e. hyperthreading). Unsurprisingly, threads are the prevailing building blocks for concurrency in most programming languages. However, concurrent programming based on threads, locks and shared state is said to be difficult and error-prone [Suto5].

5.2.1 The Implications of Shared and Mutable State

Conceptually, a thread describes a sequential flow of control, that is isolated from other activities at first glance. Unlike processes, threads share the same address space though. That implies that multiple independent threads may access the same variables and states concurrently. Even worse, sequential programming is built on the concept of mutable state, which means that multiple threads may compete for write operations, too. Multithreading is principally used with preemptive scheduling. As a result, the exact switches and interleavings between multiple threads are not known in advance. This represents a strong form of indeterminacy. Without further care, mutable state and indeterminacy introduce the strong hazard of race conditions.

A race condition occurs when two or more thread compete for access to critical section, a section that contains state shared between threads. Due to the variety of possible interleavings, the race condition may result in various inconsistent states. For instance, a thread may read stale state while another thread is already updating it. When multiple threads alter the state at the same time, either one of the changes may last and the others get lost, or even a inconsistent state affected by multiple changes may persist. Eventually, we need mechanisms to guard critical sections and enforce synchronized access.

Locking Mechanisms

The general primitives for synchronization are locks, that control access to critical sections. There are different types of locks with different behaviors and semantics. Semaphores [Dij65] are simple locks that provide a `wait` and `signal` function. Before entering a critical section or using a

shared resource, the `wait` function must be called. Once the critical section has been traversed, it is freed using `signal`. The semaphore prevents multiple threads from acquiring the semaphore at the same time by blocking other contenders in the wait call. Other semaphore implementations, so-called counting semaphores, allow a bounded number of threads to pass. Hence, a binary semaphore can be considered as a counting semaphore limited to a single active thread. Other constructs for mutual exclusion provide the concept of an ownership, which means that the critical section is temporarily possessed by a distinct thread, which is also the only instance able to unlock it later.

A more advanced construct for mutual exclusion is the monitor [Hoa74, Lam79b] that modularly protects sections using condition variables. Often, these sections have the granularity of objects, or methods/functions. The internal condition variable allows a blocking thread to yield temporarily and wait for a modification of the condition triggered by other threads. A property of a locking mechanism is reentrancy. When a lock supports reentrancy, a thread that has already obtained a certain lock can pass the lock again. This is an important property for recursive functions, that may repeatedly access critical sections. Besides counting locks, there are also locks that differentiate between different access semantics. Read/write locks allow shared access for reading threads, but exclusive access for threads that demand write access.

The Consequences of Locking

Locks allow us to serialize access to critical sections. The usage of mutual exclusions yields an atomic behavior of threads within critical sections, because its execution appears as a single operation to other waiting threads. Identifying sections of code vulnerable to race conditions and carefully placing locks around tames indeterminacy and enforces serialized access.

However, the concept of locks has introduced another danger for multi-threaded code. Improper locking may actually break the application, when obtained locks are not released or locks to acquire never become available. It is obvious that faulty locking can occur when developers must explicitly place `wait` and `signal` functions to guard sections. Higher-level abstractions like monitors often provide means to mark entire sections of code for mutual exclusion and implicitly acquire and release locks. However, they can still fall victim to locking issues. The most notorious locking issue is the so-called deadlock. It occurs when two or more threads compete for locks with cyclic dependencies. In the simplest scenario, two threads both own a separate lock, but additionally need to acquire the lock of the other thread. As no thread can advance without acquiring a second lock, both threads are blocked and cannot continue.

Other locking issues include livelocks and lock starvations. Similar to deadlocks, livelocks prevent threads to continue. However, threads are not blocked in a livelock situation. Instead, they steadily change states in response to other state changes of other threads involved, which in turn also change states. In an example scenario, two threads must acquire two resources in order to continue. When they cannot obtain both resources, they will return the first one and retry to obtain both. The livelock appears when two threads simultaneously start to claim a first

resource, then restart again. Livelocks can be considered as special case of starvation. It generally describes the scenario when a thread is repeatedly unable to acquire a lock or resource, because other greedy threads are constantly claiming it.

While some starvation issues such as livelocks might be handled at runtime using random backoffs for retries, potential locking issues are generally very difficult to detect due to non-determinism. The risk of a deadlock increases when multiple locks with fine granularities are used. Accordingly, the use of coarse locks is often recommended in order to avoid deadlocks. Identifying large critical sections and guarding them by locks not just ensures serialized access. In fact, coarse locks result in an eventually sequential execution of threads. This is contrary to our prior goal of increased parallelism.

For concurrency with true hardware parallelism, we need to choose very fine locking granularities, that enable multiple threads to continue in independent critical sections. Many small critical sections not just increase the overhead of locking management, since locking is not free in terms of management resources. Yet again, the extensive use of locks emphasises the risk of the aforementioned dangers. It becomes clear that it is not easy to pick the right locking granularity.

Besides the issues of livelocks, deadlocks and starvations, there is also another difficulty with locks in practice. Given multiple pieces of code with critical sections protected by locks, we cannot guarantee that the composition of these pieces of code does not yield a deadlock. Essentially, we cannot compose thread-safe implementations without the risk of new locking issues. This is especially significant for larger code fragments such as framework components or modules. Locking issues may be tackled by resolute development policies, that strictly govern the usage, obtain order and conditions of locks. However, such policies cannot be enforced programmatically. Moreover, when external or closed-source components are used, it becomes impossible to ensure correct locking.

Nevertheless, concurrent programming based on threads, shared state and locking is still prevailing and available in most languages. It is important to recognize that this approach represents a low-level concept towards concurrency. It is closer to the bare metal than the other concepts we will see soon. However, all of these concepts still use threads under the hood.

5.2.2 Case Study: Concurrency in Java

The Java programming language has been providing thread-based concurrency from the beginning of its existence. It implements Mesa monitors [Lam79b] for locking and mutual exclusion, and provides several synchronization primitives as part of the language core. The concurrency behavior is defined in the Java Language Specification [Gos12] which describes the Java Memory Model (JMM) in detail. Java's consistency is based on a happens-before order and the notion of an implicit memory barrier. However, it does not provide sequential consistency for threads, as many developers erroneously assume. In fact, the JMM resembles symmetric multi-processing, where multiple CPUs have their own cache, but share a common maim memory. When CPUs access memory, they refresh their cache and eventually flush changes. In a metaphorical sense,

Java threads resemble the CPUs, the main memory is the shared memory between threads and the CPU caches are thread local copies of data. The procedure of flushing or refreshing represents the traversal of a so-called memory barrier. Besides the aforementioned ordering, JMM defines also which operations cannot be interrupted by other threads (atomicity) and when changes have to be propagated to other threads (visibility), based on the memory barrier. For instance, starting and ending a thread or using synchronization primitives touches the memory barrier, but also access to several variables with specific traits (see below).

The `synchronized` keyword allows to guard an entire method or a distinct code block, using the callee resp. a given object as monitor object. Java monitors are reentrant and recursive calls are supported. Furthermore, every Java `Object` can be used as a monitor and hence provides means for condition signaling. The method `wait()` blocks the thread holding the monitor and releases the monitor in order to allow other threads to proceed and change the condition. In this case the other threads can use `notify()` and `notifyAll()` for signaling a waiting thread.

The `volatile` keyword circumvents the thread-local copy of a variable and enforces a fresh copy from the shared memory on each access. It can only be used for single atomic operations. For instance, incrementing a value (multiple operations) is not atomic. The `final` keyword makes a variable immutable. The benefit of immutable values for concurrency is obviating the need for refreshing values, as they cannot be changed anymore. It is recommended always to set fields to final, unless there is a reason not to do so [Bloo8]. Furthermore, Java provides a set of atomic entities (`java.util.concurrent.atomic`), similar to volatile variables. However, these entities are objects, ensure atomicity of all operations and use very efficient mechanisms internally such as compare and swap.

Activities are represented by the `Thread` class, that provides methods for thread handling such as `start()`. This class also has several coordination methods such as `resume()` and `stop()`, that are unfortunately broken and should not be used [Bloo8]. The `Runnable` interface abstracts from the Thread class and only possesses a `run()` method, the method eventually executed by a thread once started.

While this is the foundation of concurrent programming in Java, several higher-level abstractions have been introduced, starting with Java 5. The main reason was to to facilitate the development of concurrent applications. Explicit locks (`java.util.concurrent.locks`) provide more extensive locking operations (e.g. read-write locks) than the implicit monitior-based locks of `synchronized`. Concurrent collections, such as `ConcurrentMap` or `BlockingQueue`, extend existing collections and provide thread-safe operations as well as operations for coordinated access. Another abstraction is provided by `Runnable`, `Callable` and the `Executor` framework. Essentially, these classes decouple tasks to be executed from the actual entities that execute them. In combination with thread pool entities (e.g. `ExecutorService`), this is a very helpful abstraction for many concurrent applications. `Future`s allow to asynchronously execute a `Callable` in another thread, immediately returning a proxy object to the eventual result. For more complex coordinations between threads, several high-level coordination primitives have been supplied. This includes primitives such as an explicit counting `Semaphore`, a `CountDownLatch` (a barrier

triggered by a countdown) and a `CyclicBarrier` (a barrier point for recurring coordination of threads). In Java 7, the fork/join framework [Lea00] has been introduced. This framework aims for easy parallelization of computationally heavy tasks by spawning subtasks and using divide-and-conquer strategies. It provides implicit task coordination and employs work-stealing.

Listing 5.1 shows an exemplary web application, written in Java, and using jetty[1] and Java Servlets[2]. On startup, the `CountingRequestHandler` gets instantiated a single time. Requests are internally handled in a threadpool, so concurrent requests may trigger the simultaneous invocation of the handle() method of `CountingRequestHandler`. The shared variable `count` is accessed by each thread and must be hence protected, using a synchronized block. This demonstrates the usage of monitors (in this specific case, the usage of the `AtomicLong` class would represent a more elegant and performant solution).

5.2.3 Multithreading and Locks for Concurrent Application Logic

Multi-threaded application servers assign a dedicated thread for each application request to handle. As long as there is no coordination needed between other threads, this programming model is very simple. The isolated view makes it easy to program the request logic as a sequence of operations. Also, when the request mainly executes CPU-bound operations, this approach is a valid choice. When the request logic contains I/O-bound operations, latency is generally hidden inside the server, as there are multiple requests to handle concurrently. However, this does not speed up request handling for a single request, it only increases general throughput of the server. For reducing the actual latency of a single request, we need to further parallelize operations of a single request. Additional threads help executing more work at the same time, as long as operations are independent and thus can run in parallel. The more operations are actually I/O-bound, the more we run into the same problem as seen in chapter 4. Using more threads in order to parallelize I/O yields issues due to heavy context switching and high memory consumption. In our architecture, the services are separated components that are accessed via the network, which result in a strong focus on I/O-bound operations. In essence, latency can be reduced by parallelizing work, but this works generally better for CPU-bound operations. For I/O-bound operations, this approach does not scale well.

Concerning coordination between requests, we have seen that locking, supplemented with conditional variables, can be used to coordinate threads. However, the difficulties of locking and the strong nondeterminism of incoming requests in an application server makes it rather difficult to implement completely correct inter-request coordination patterns. Instead, it is more advisable to rely on external pub/sub message components (e.g. redis), although this still blocks threads. Similarly, it is recommended to share state between requests using external facilities such as key/value stores, in order to circumvent explicit locking inside the application server.

1 http://jetty.codehaus.org/jetty/
2 http://www.oracle.com/technetwork/java/javaee/servlet/index.html

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.eclipse.jetty.server.Request;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.server.handler.AbstractHandler;

public class CountingRequestHandler extends AbstractHandler {

  //Variable for counting requests handled so far
  private long count = 0;

  public void handle(String target, Request baseRequest,
      HttpServletRequest request, HttpServletResponse response)
      throws IOException, ServletException {

    response.setContentType("text/plain");
    response.setStatus(200);
    baseRequest.setHandled(true);

    final long current;
    //Access and modification of a variable shared between threads.
    synchronized (this) {
      current = ++count;
    }

    response.getWriter().println(""+current);
  }

  public static void main(String[] args) throws Exception {
    Server server = new Server(8080);
    server.setHandler(new CountingRequestHandler());

    server.start();
    server.join();
  }
}
```

Listing 5.1: A minimalistic, concurrent web application written in Java that returns the number of requests handled so far.

Another advantage of this approach is the possibility to transparently scale out by instantiating new application servers.

5.3 Concurrency via Software Transactional Memory

We have seen that lock-based concurrency has several drawbacks. Indeterminacy and shared state requires a protection from race conditions. The concept of locks holds the developer responsible for guarding critical sections by explicitly placing locks. In turn, this may yield unpredictable locking issues at runtime due to lock orders and indeterminacy. Also composability of concurrent code is not guaranteed when locks are used.

We now examine an alternative approach, that is still built on the concept of shared state and locks, but does not rely on the developer's reasoning for correct locking. Instead, locking becomes part of the underlying runtime environment, and the programming language provides higher abstractions for concurrent sections of code.

5.3.1 Transactional Memory

A lock-free alternative for shared state concurrency is Transactional Memory (TM). It goes back to a well-known concept in computer science, transactions. The idea of concurrent, yet isolated operations is an established concept for database systems [Helo7]. TM takes up this idea and applies it to shared state concurrency [Kni86, Her93]. While database transactions read and write on database rows, TM transactions read and write on state shared with other threads. The concept of TM can be implemented in various ways. Hardware Transactional Memory (HTM) provides an hardware implementation of TM, that extends CPU architectures by transactional components such as transactional caches and an extended instruction set. Software Transactional Memory (STM) does not require any hardware changes and supports transaction handling entirely in software. Hybrid TM is essentially an STM implementation that takes advantage of progressing hardware support for TM. Due to high development and implementation costs of HTM, TM is primarily available in the form of STM. However, some believe that hybrid models may eventually appear, once this programming model has been established [Caso8].

Traditional transactions (see chapter 6) provide several guarantees, namely atomicity, consistency, isolation and durability. Hence, transactions appear as single operations that do not yield inconsistent state while running but not having committed or aborted yet. They also do not interfere with other running transactions and their outcome is always persisted. Note that this kind of durability differs from transactions for database systems and TM. The latter only keeps the transaction outcome in memory, but does not recover state from application crashes. As a consequence of these properties, transactions are serializable. The outcome of transactions can be reproduced by an equivalent sequential execution of seemingly atomic operations. Concurrency control for transactions can either be pessimistic or optimistic. Pessimistic concurrency control forces conservative locking of resources and results in low transaction throughput. Optimistic

concurrency control delays the integrity checks of a transaction to its end. In case of a conflict, the transaction is aborted and gets restarted. When transactions are not long-running and do not conflict too often, optimistic concurrency control provides a very good performance with a negligible overhead of retries.

5.3.2 Software Transactional Memory

We will now solely focus on STM, as there are already several implementations available. Optimistic concurrency control is preferred by existing STM implementations. In order to integrate STM concepts into a language and implement an underlying STM runtime, it is important to realize what constructs are necessary on language level. On one hand, we need a way to label sections of code as transactional. On the other hand, we might want to differentiate variables and resources that are shared between threads and need transactional call semantics and non-transactional, thread-local variables. Otherwise, any read or write operation would result in a transaction.

Once a transaction has been started at runtime, the underlying implementation starts to keep a read set and a write set [Her93]. Both sets contain all variables and states that the transaction has read or altered. This is necessary for a later integrity check before committing. Also, as long as the transaction is pending, changes are not applied to the actual share variables, but on thread-local copies, often in form of a transaction log. Once the transaction has been verified as not conflicting, all of its changes are then flushed to the actual shared states in an atomic step. While this often contains some forms of locking, this behavior is entirely transparent for the developer. In order to detect conflicting transactions, the STM implementation compares the read and write sets of a transaction with the actual states before committing. When another transaction has already altered a state and has committed successfully, the STM detects the discrepancy and aborts the transaction. Instead, the old read and write sets get discarded and refreshed, and the transactions restarts. To some extent, starvation situations can still occur, especially when a long-running transaction is steadily outpaced by other transactions that successfully commit first. Apart from that, STM provides mutual exclusion without explicit locking, and without the danger of the aforementioned locking issues so far.

In order to represent a valuable concurrency model, additional features are still needed. We have seen that lock-based multithreading lacks support for composability. Also, we need mechanisms to coordinate different threads using STM. Both requirements have been addressed in an extended STM model [Har08]. The granularity of transactions allows to glue together different transactional operations, yielding a new, again transactional composite operation. More advanced compositions can be implemented using operators such as `retry` and `orElse`. The former operator is based on a concept similar to that of condition variables for monitors. When a running transaction checks a condition containing a state that differs from the expected value, the transaction can "yield" by calling `retry`. Thanks to the read set, the underlying runtime detects which variables have been accessed so far. Once one of these variables has been altered in other transactions, the

runtime resumes the prior transaction and checks if the condition evaluates to true. The `orElse` operator allows to compose two or more transactions. When the the first transaction calls yields via `retry`, the next transaction is executed instead. In essence, these operators introduce the concept of blocking coordinations into the transactional model. Hence, transactions can now wait for events to occur, or include alternative control flow in a single transaction behavior.

There are important limitations of STM. As TM transactions are limited to memory operations, they can only be used when coordinating access to shared state, but not to external resources. Furthermore, the transactional character requires operations to be irrevocable, so transactions must not have any side effects apart from modifications of shared state. For example, the usage of I/O operations inside transactions is disallowed.

Furthermore, the length of transactions and the ratio of conflicting transactions have a lasting effect on the performance of STM deployments. The longer transactions take to execute, the more likely they cause conflicts and must be aborted, at least in case of many contending transactions.

However, STM provides a beneficial extension of the traditional concurrency model of shared state and threads that evades the burden of locking. It allows to compose concurrent operations without the danger of deadlocks. However, contention and starvation can still occur in a STM system. The former is often the result of many simultaneous transactions altering the same values. The latter might become apparent when a very lengthy transaction continuously competes with multiple short transactions.

When implemented with optimistic concurrency control, STM provides reasonable performance, as long as there are not many concurrent and conflicting write operations.

5.3.3 The Transactional Memory / Garbage Collection Analogy

The idea of TM and the STM approach for lock-based programming are still controversial. Critics argue that STM continuously faces several challenges [Cas08], and that's why it is still primarily of academical interest so far. For instance, it is still unclear how to handle transactional and nontransactional access to the same variable, and how to privatize variables (i.e. switching from transactional to thread-local access). Also, the drawbacks of requiring side-effect free code raises the question, how to incorporate code that cannot be defined as a transaction into transactional operations. Still the most prominent argument against STM is the performance overhead induced by the software-based implementation of transaction handling.

Proponents of STM counter that the performance of recent STM systems has vastly increased and STM already represents a robust solution [Dra11]. Various implementations also came up with different approach to privatization. Last but not least, the continuing success of Clojure[1] testify the maturity of newer STM implementations. Clojure is the first programming language that has a STM as first-class, built-in concurrency concept. Prior to Clojure, STM implementations were mainly found as extensions to Concurrent Haskell, based on special monads.

1 `http://clojure.org/`

Probably the most interesting notion in this argument around TM is the analogy to garbage collection [Gro07]. While garbage collection addresses managed references, TM addresses managed state. Both concepts operate on the memory at runtime and take difficult work out of the hands of application developers. Compared to TM, very similar objections have been raised against garbage collection, when it has been suggested for the first time. Also, the first garbage collectors suffered from observable performance overheads compared to manual memory management. However, garbage collectors have been vastly improved over time are now an integral part of many high-level languages. As the model of shared memory will continue to prevail in the near future, time will tell if the analogy goes on and TM will establish itself as a reasonable high-level abstraction for shared state concurrency.

5.3.4 Case Study: Concurrency in Clojure

Clojure[1] is a programming language that runs on the JVM and is heavily influenced by Lisp. It has a strong focus on functional programming concepts. Another defining feature of Clojure is its elaborate approach towards concurrency. In fact, concurrent programming has been one of the main reasons for developing Clojure in the first place [Hic08]. Like Scala, Clojure builds on Java and internally uses Java concurrency primitives. Clojure provides a very strong immutability concept combined with asynchronous agents and a mature implementation of STM.

In order to understand the cooperation between those concepts, it is necessary to elaborate the notions of identity, state, references and values for Clojure in detail. A value is an immutable piece of data that does never change. This is even true for imperative programming to some extent. For instance, we don't directly change the value (i.e. number) of a numeric variable, we rather assign another value to the variable instead. This does not affect the old value though. Object-oriented programming obfuscates this concept by unifying identity and state. In Clojure, an identity is "a stable logical entity associated with a series of different values over time"[2]. In other words, an identity is an entity with a mutable association to a value, an the state of an identity is captured by its value at a given time. A reference points to an identity, which in turn points to a value, depending on the current state. State changes are reassignments of identities to other values.

While this is self-explanatory for values such as numbers, Clojure applies this principle to data structures as well. When a list is changed by adding a new element, the new value is the old list appended with the new element. However, the old list still remains unchanged. For supporting this concept for non-primitive data types such as lists and maps, Clojure makes use of so-called persistent data structures [Oka96]. In this context, persistent does not denote durable storage. Instead, persistent data structures preserve their history. Efficient implementations that hold multiple versions of a data structure without redundant values represent the main challenge for persistent data structures. This indirection is an important property for the concurrency concept

1 http://clojure.org/
2 http://clojure.org/state

of Clojure and preserves the immutability of values.

The runtime system of Clojure supports state changes based on values and identities automatically. Therefore, Clojure provides four different types of references to mutable state, that have different impacts, as shown in table 5.1. The `var` reference primitive resembles traditional variables of imperative programming languages that can be reassigned to other values. However, `var`s are only thread-local, and cannot be accessed by other threads. Consequentially, state is mutable, but not shared.

The `atom` reference primitive is very similar to the atomic entities of Java. They allow to manage shared, synchronous, independent state. The state of an `atom` can be accessed by explicit dereferencing. For changing the value, there are three operations, namely `reset` (setting a new value), `swap` (applying a modification function) and `compare-and-set` (lower-level variant).

The `ref` primitive defines references that can be accessed for a read operation by using dereferencing. Modifying operations can only be executed as part of a STM transaction. The STM implementation of Clojure uses Multiversion Concurrency Control (MVCC) [Ber81, Ree78], a concurrency control mechanism based on timestamps. The `dosync` function is used for encapsulate transactional operations and all function calls in its function body run in the same transaction. For operating on `ref`s inside a transaction, there are different functions. `ref-set` is used for directly setting the `ref` to a new value. The `alter` operation applies a function which implements the exchange of state of the `ref`. The `commute` operations works the same way like `alter`, but implies that the modifying function is commutative, effectively allowing more concurrency internally (using in-transaction values). Finally, `ensure` prevents other transactions from setting an in-transaction value for the `ref`. This avoids write skews: multiple transactions read overlapping data sets, but make disjoint modifications without seeing the changes of other transactions. As already mentioned, transactions should not contain operations with side effects, such as I/O operations. That's because the internal STM implementation may abort and retry transactions.

The last primitive is `agent`, providing independent, asynchronous and shared access to mutable state. Agents isolate state that can be dereferenced by threads for read access. Threads can also send actions (i.e. modifying functions) to an agent. The agent then executes incoming actions sequentially. Execution happens asynchronously in regard to the sender of the action, and execution is always guaranteed to run single-threaded per `agent`. The agent concept is different from the idea of an actor, as we will see soon. An agent executes incoming functions on its

	synchronous	**asynchronous**
coordinated	`ref`	
independent	`atom`	`agent`
thread-local	`var`	

Table 5.1: Clojure primitives for handling mutable state.

internal state. Hence, the sent functions define a behavior. An actor provide its own internal behavior and waits for handling incoming immutable messages.

Clojure forces developers to explicitly label mutable state. Otherwise, state cannot be modified at all. Only this makes Clojure applications more robust, as it prevents accidental and unintended mutability. It requires the developers to pick an appropriate concurrency primitive for state handling.

Avout[1] is an external contribution to Clojure that provides a distributed MVCC STM implementation based on Apache ZooKeeper[2] for coordination. It enables the usage of `atom` and `ref` primitives between multiple (remote) JVM instances.

Listing 5.2 provides a Clojure-based solution to our previous web application example. The solution takes advantage of the noir web framework[3], which in turn uses the jetty web server. The minimalistic application defines a `counter` of type `ref` and a registers a function for request handling. On each request, this functions executes an STM transaction within the `dosync` block. In this case, the `commute` operation is used, which `inc`rements the `counter` value transactionally. Usually, the `alter` method is used instead of `commute`. However, the increment operation is commutative, hence we might speed up the transaction execution when using `commute`. After the transaction has finished, the value of `counter` is dereferenced (`@`), converted into a `str`ing and returned as response. We deliberately dereference the value of the `counter` `ref` outside rather than inside the actual transaction, in order to demonstrate the possibility non-transactional read access to `ref`s (this yields a short window in which the value might have already been changed by another request). Like in the previous Java example, it would be more elegant to use a Clojure `atom` instead, as the counter is the only variable to be modified.

```
(ns counting.server
  (:use noir.core)
  (:require [noir.server :as server]))

(def counter (ref 0))

(defpage "/" []
  (dosync (commute counter inc))
    (str @counter))

(server/start 8080)
```

Listing 5.2: A minimalistic, concurrent web application written in Clojure that returns the number of requests handled so far.

1 http://avout.io/
2 http://zookeeper.apache.org/
3 http://webnoir.org/

5.3.5 STM for Concurrent Application Logic

Like in the previous lock-based approach, application servers using STM also map requests to threads. We have seen that this approach becomes increasingly inadequate, when I/O-bound operations dominate. STM does not provide a solution to this issue, in fact, it disallows I/O operations inside transactions at all. However, STM can support concurrent application logic, when state is shared in an application server. Depending on the type of application, application state may be sharded and isolated to several distinct servers (e.g. multiplayer games with small parties hosted on a distinct server), or it must be available for all application servers (e.g. instant message notifications in social web applications). In the latter case, distributed STM variants allow for distribution aspects. When the STM implementation provides mechanisms that are similar to condition variables, coordination between threads as part of transactions is also supported.

As a result, STM renders shared state inside application servers more manageable, thanks to the absence of explicit locking, but does not solve I/O-bound parallelization issues.

5.4 Actor-based Concurrency

The concurrency models we have considered so far have the notion of shared state in common. Shared state can be accessed by multiple threads at the same time and must thus be protected, either by locking or by using transactions. Both, mutability and sharing of state are not just inherent for these models, they are also inherent for the complexities. We now have a look at an entirely different approach that bans the notion of shared state altogether. State is still mutable, however it is exclusively coupled to single entities that are allowed to alter it, so-called actors.

5.4.1 The Actor Model

The actor model has its theoretical roots in concurrency modelling [Hew73] and message passing concepts [Hoa78]. The fundamental idea of the actor model is to use actors as concurrent primitives that can act upon receiving messages in different ways:

1. Send a finite number of messages to other actors.
2. Spawn a finite number of new actors.
3. Change its own internal behavior, taking effect when the next incoming message is handled.

For communication, the actor model uses asynchronous message passing. In particular, it does not use any intermediate entities such as channels. Instead, each actor possesses a mailbox and can be addressed. These addresses are not to be confused with identities, and each actor can have no, one or multiple addresses. When an actor sends a message, it must know the address of the recipient. In addition, actors are allowed to send messages to themselves, which they will receive and handle later in a future step. Note that the mapping of addresses and actors is not part of the conceptual model (although it is a feature of implementations).

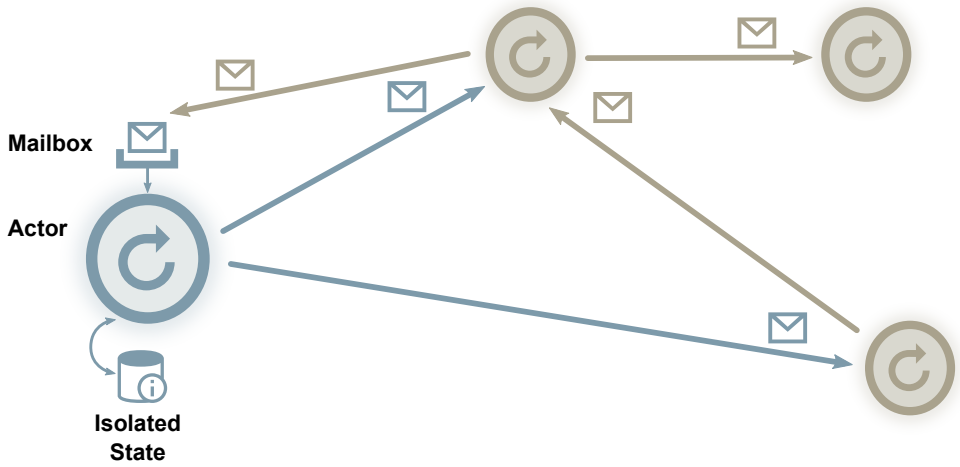

Figure 5.3: An example network of several actors. Each actor has its own mailbox and isolated state. Based on its designated behavior, the actor responds to incoming messages by send new messages, spawn new actors and/or changing its future behavior.

Messages are sent asynchronously and can take arbitrarily long to eventually arrive in the mailbox of the receiver. Also, the actor models makes no guarantees on the ordering of messages. Queuing and dequeuing of messages in a mailbox are atomic operations, so there cannot be a race condition. An actor processes incoming messages from his mailbox sequentially using the aforementioned possibilities to react. The third possibility, changing its own internal behavior, eventually allows to deal with mutable state. However, the new behavior is only applied after the current message has been handled. Thus, every message handling run still represents a side-effect free operation from a conceptual perspectice. The actor model can be used for modelling inherently concurrent systems, as each actor is entirely independent of any other instances. There is no shared state and the interaction between actors is purely based on asynchronous messages, as shown in figure 5.3.

5.4.2 Actor Implementations for Concurrent Programming

Besides a theoretical model for concurrent systems, the idea of actors also represents the blueprint for a concurrent programming model. Several conceptual implementations have been considered [Agh90], ranging from strictly functional adaptions to extensions of the object-oriented paradigm [Gue07]. The first, fairly popular programming language that has incorporated the actor model for concurrency was Erlang [Vin07]. The actor model has recently become increasingly popular and finds its way into many new programming languages, often as first-class language concept. In many other languages, the actor model is available using third-party libraries that build on top of conventional multithreading.

When implementing the actor model, it is important to adhere to the set of rules defined by the original idea. First and foremost, actors must not share any state. This disallows actors to pass references, pointers or any other kind of shared data as part of a message. Only immutable data and addresses (i.e. "names") of actors should be sent. Message passing between actors is often enriched with a few more guarantees compared to the entirely best-effort style. Most implementations ensure that two messages sent from one actor to another maintain their order at arrival. Messaging is always asynchronous and the interleaving of incoming messages sent by multiple actors is indeterminate.

For message handling, most implementations provide pattern matching. This enables the developer to distinguish between different types of messages and supply different handling code associated to the type of message. While receiving messages is a blocking operation from the actor's perspective, sending new messages is always non-blocking. Some implementations also provide selective receive semantics. Depending on its behavior, an actor may then wait for a specific message in the mailbox and temporarily defer others.

The underlying runtime platform must allocate possibly huge numbers of actors to restricted numbers of CPU cores and resources and schedule the execution of actors with pending messages. Most systems employ a principally lock-free implementation, as atomic behavior is only required for the mailbox operations. For instance, the Erlang virtual machine starts a single process and spawns a pool of threads based on the number of cores available [Lar08]. The internal scheduler organizes actors in process queues and works preemptively. When a running actor has handled a message or has executed a certain number of reductions (i.e. function calls), the scheduler switches to the next actor ready to run. For I/O operations, the Erlang virtual machine spawns decoupled operating system threads. The reduction limits and background I/O operations promote fairness and liveness, because no actor can bind CPU time for a longer period.

Many implementations of the actor model provide additional features that are also ramifications of the actor model, namely distribution support and fault tolerance. Concurrency concepts normally target single machines. The actor model does not postulate many guarantees for messaging. Asynchronous, unbounded messaging in fact resembles network-based communication. The isolation of states to actors does not require shared access between multiple actor instances. Actors are designated using addresses, that can easily provide location transparency. The actor model is inherently parallel, thus it is very easy to extend implementations of the actor model to support distributed deployments. For instance, distributed Erlang systems make use of multiple nodes running an Erlang virtual machine and transparently provide distributed messages passing.

Conventional, thread-based concurrency gives fault tolerance a hard fight. Nondeterminism and unpredictable scheduling combined with shared state and locking requires very complex strategies for replication and snapshotting. The actor model comes up with other primitives that makes replication much easier. Isolated states and incoming messages queued in actors' mailboxes are very similar to snapshots and logs. Message handling of an actor is single-threaded and provides implicit yielding points. Erlang embraces a "let is crash" philosophy [Arm07]. The

isolated, shared nothing trait of actors allows a single actor to fail without affecting any other actors. Furthermore, the actor model itself can be used for fault tolerance, by spawning hierarchy trees of actors for supervision. Once an actor crashes, the supervising actor receives a messages and can react. A supervisor might restart the actor, stop other actors or escalate by sending an error message to its own supervisor.

Actors have an isolating impact on state and effectively prevent shared mutabale state. Also, no locks are necessary for concurrent programming. However, concurrency issues like deadlocks and race conditions are still not entirely expelled in this programming model, as they can be reintroduced by incorrect applications. Two actors waiting for a message from each other represent a cyclic dependency. In practice, the impending deadlock can be prevented by using timeouts. The arbitrary ordering of messages send by actors might be interpreted as a traditional race condition by some developers. However, it is a characteristic property of the actor model testifying asynchrony. Hence, these developers ignore fundamental ideas of the actor model and the resulting "race condition" is actually a manifestations of inappropriate application design.

5.4.3 Programming with Actors

The actor model for concurrency is very different than thread-based concurrency with locks or STM. Isolated mutable state and asynchronous messaging yield other programming patterns that threads do.

First of all, it is important to understand that actors represent very lightweight primitives compared to threads. They can be spawned and destroyed with minimal overhead. Thus, it is totally feasible to create and use large numbers of instances in parallel. Actors can also execute arbitrarily complex computations in response to a message. Actors can send messages to themselves, which allows messaging patterns that recreate recursion. Furthermore, actors can send messages to other actors known by their address, so an actor-based program is essentially a highly dynamic network of actors (a directed graph). As a result, existing message-based patterns for application integration [Hoho3] provide a comprehensive set of patterns that can be used for concurrent programming with actors as well. This includes popular messaging patterns for routing, filtering, transformation and composition.

Isolating mutable state and enforcing immutable messages guarantees implicit synchronization. However, the concept of asynchronous messaging and no global state challenges coordination. An application may require consensus or a concerted view of state between multiple actors. When multiple actors must be strictly orchestrated in order to provide a distinct application function, correct messaging can become very demanding. Thus, many implementations provide higher-level abstractions that implement low-level coordination protocols based on complex message flows, but hide the internal complexity from the developer. For Erlang, OTP is a standard library that contains a rich set of abstractions, generic protocol implementations and behaviors.

Another common approach is the transactor. For example, multiple actors may require to modify their internal state in a coordinated manner. A transactor, which is a dedicated actor

for coordinating transactional operations of multiple actors, can help in this situation by providing abstract transaction logic. Some transactors also apply STM concepts for transactional behavior [Les09, Les11].

5.4.4 Case Study: Concurrency in Scala

Scala[1] is a general purpose, object-functional language that runs on the JVM. It interoperates with Java, but provides enhanced expressiveness, advanced programming concepts and many features of functional programming. For concurrency, Scala implements an actor-based concurrency model and supports explicit immutability of values. However, Scala applications can also fall back to concurrency primitives of the Java programming language.

While Erlang spawns multiple low-level threads and implements a custom scheduler for running actors, Scala is somehow caught by the multithreading implications of the JVM. Furthermore, the actor implementation of Scala is not part of the language core, but part of its standard library. This means that actor library itself is implemented in Scala. One initial challenge of Scala actors has been introduced by the constraints of multithreading in the JVM. The number of possible threads is limited, there is no cooperative scheduling available and threads are conceptually less leightweight than actors are supposed to be. As a result, Scala provides a single concept of an actor, but two different mechanisms for message handling [Hal06, Hal08].

Thread-based Actors When the `receive` primitive is used, the actor is internally backed by a dedicated thread. This obviously limits scalability and requires the thread to suspend and block when waiting for new messages.

Event-driven Actors The `react` primitive allows and event-driven execution strategy, which does not directly couple actors to threads. Instead, a thread pool can be used for a number of actors. This approach uses a continuation closure to encapsulate the actor and its state. However, this mechanism has several limitations and obscures the control flow [Hal08]. Conceptually, this implementation is very similar to an event loop backed by a threadpool. Actors represent event handlers and messages resemble events.

Generally, `react` should be preferred, as it does not couple each actor to a dedicated thread. The `react` primitive thus yields better scalability results.

The syntax of Scala actors for messaging follows the Erlang style. Messages are supposed to be immutable values, but this is not enforced so far. Often, case classes are used, a special type of wrapper class. They are especially helpful when pattern matching is used to determine the type of message on arrival. Scala also supports the distribution of actors, by supplying remote actors, that communicate over TCP/IP and rely on Java serialization.

1 `http://www.scala-lang.org/`

```
package actors

import akka.actor.Actor
import Actor._

class CoutingActor extends Actor {

  var count = 0;

  def receive = {
    case "visit" =>
      count = count+1
      sender ! ""+count
  }
}
```

Listing 5.3: An actor in scala, based on the akka actor implementation. The actors encapsulates a counter state, and responds to each "visit" message by returning the number of overall visits counted so far.

Listing 5.3 and 5.4 illustrate an actor-based solution to our prior exemplary web application. The solution is written in Scala and uses the Play web application framework[1]. The framework does not use regular Scala actors, but actor implementations provided by the akka library[2]. As shown in listing 5.4, the application starts a single actor and registers a method for handling requests. This method sends an asynchronous `"visit"` message to the actor. By using the `?` operator, a `Future` is returned that represents the eventual reply. If no timeout occurs, the actor reply is then used as response body of the `SimpleResult`, once available. The internals of `CoutingActor`, as shown in listing 5.3, are very simple. The actor provides a single counter variable and responds to "visit" messages by incrementing the value and sending it back to the original sender.

5.4.5 Actors for Concurrent Application Logic

When the application servers use the actor model, each incoming request represents a new actor. For parallelizing request operations, the actor spawns new actors and assigns work via messages. This enables parallel I/O-bound operations as well as parallel computations. Often, the flow of a single request represents a more or less complex message flow between multiple actors, using messaging patterns such as scatter/gather, router, enricher or aggregator [Hoh03].

However, implementing request logic using actors differs clearly from sequential request logic implementations. The necessary coordination of multiple actors and the less apparent flow of execution due to asynchronous messaging provides an arguably less comfortable, but more

1 http://www.playframework.org/
2 http://www.akka.io/

```
package controllers

import akka.util.Timeout
import akka.pattern.ask
import akka.actor._
import akka.util.duration._
import actors.CoutingActor
import play.api._
import play.api.mvc._
import play.api.libs.concurrent._
import play.api.libs.iteratee.Enumerator

object Application extends Controller {

  val system = ActorSystem("counter")
  val actor = system.actorOf(Props[CoutingActor])

  def index = Action {
    AsyncResult {
      implicit val timeout= Timeout(5.seconds)
      (actor ? "visit").mapTo[String].asPromise.map { result =>
        SimpleResult(
          header = ResponseHeader(200, Map(CONTENT_TYPE -> "text/plain")),
          body = Enumerator(result)
        )
      }
    }
  }
}
```

Listing 5.4: A minimalistic, concurrent web application written in Scala that returns the number of requests handled so far, using the Play web framework.

realistic abstraction towards concurrency

Another feature of the actor model is the possibility to scale the actor system as a whole by adding new machines. For instance, Erlang enables virtual machines to spawn a distributed system. In this case, remote actors can hold isolated application state, but accessible via messaging for all other actors of the entire system.

5.5 Event-driven Concurrency

We have already got to know the concept of event-driven architectures in chapter 4. Although event-driven programming does not represent a distinct concurrency model per se, the concepts of event loops and event handlers and their implementations have strong implications on concurrent programming. Events and event handlers are often confused with messages and actors, so it is important to point out that both concepts yield similar implementation artifacts. However, they

do not have the exactly same conceptual idea in common. Actor-based systems implement the actor model with all of its characteristics and constraints. Event-driven systems merely use events and event handling as building blocks and get rid of call stacks. We have now a look at the original idea of event-driven architectures "in the large", and examine the event-driven programming model afterwards.

5.5.1 Event-driven Architectures

Whether in small application components or in large distributed architectures, the concept of event-driven architectures [Hoho6] yields a specific type of execution flow. Conventional systems are built around the concept of a call stack that bears on several assumptions. Whenever a caller invokes a method, it waits for the method to return, perhaps yielding a return value. Finally, the caller continues with his next operation, after his context has been restored. The invoked method in turn might have executed other methods on its own behalf. Coordination, continuation and context are thus inherent features of the call stack. The imperative model assumes a sequential execution order as a distinct series of maybe nested invocations. A caller knows in advance which methods are available and which services they expose. A program is basically a path of executions of instructions and method invocations. Hence, a call stack is very formative for programming concepts and it so pervasive, that many developers take it for granted.

Surprisingly, event-driven architectures reject the concept of a call stack and consequently lose its inherent features. Instead, these architectures promote more expressive interaction styles beyond call/return and a looser coupling of entities. Basic primitives of this concept are events. Events occur through external stimuli, or they are emitted by internal entities of the system. Events can then be consumed by other entities. This not just decouples callers and callees, it also obviates the need for the caller to know who is responsible for handling the invocation resp. event. This notion obviously causes a shift of responsibilities, but allows more versatile compositions and flows. The event-driven architecture is also inherently asynchronous, as there is neither a strong coupling between event producers and event consumers, nor a synchronous event exchange.

In essence, the event-driven model represents an approach for designing composable and loosely coupled systems with expressive interaction mechanisms, but without a call stack. Event-driven architectures do not represent or nominate a certain concurrency model. Unsurprisingly, thread-based event loops and event handlers are often used for implementations of event-driven architectures, next to message passing approaches.

5.5.2 Single-threaded Event-driven Frameworks

Several platforms and frameworks use event-driven architectures based on event loops and event handlers for the implementation of scalable network services and high-performance web

applications. Popular solutions include node.js[1], Twisted Python[2], EventMachine (Ruby)[3] and POE (Perl)[4].

The programming languages used by these systems still provide a call stack. However, these systems do not use the call stack for transporting state and context between event emitters and event handlers.

While it is generally possible to apply multithreading [Zelo3], most of the aforementioned frameworks and platforms rely on a single-threaded execution model. In this case, there is a single event loop and a single event queue. Single-threaded execution makes concurrency reasoning very easy, as there is no concurrent access on states and thus no need for locks. When single-threading is combined with asynchronous, non-blocking I/O operations (see chapter 4), an application can still perform very well using a single CPU core, as long as most operations are I/O-bound. Most web applications built around databases operations are indeed I/O-bound, and computationally heavy tasks can still be outsourced to external processes. When applications are designed in a shared-nothing style, multiple cores can be utilized by spawning multiple application instances.

In chapter 4, we have seen that single-threaded event-driven programming does not suffer from context switching overheads and represents a sweet spot, when cooperative scheduling is combined with automatic stack management. We have also seen previously that the missing call stack in event-driven architectures relinquishes free coordination, continuation and context.

The event-driven programming style of these frameworks provide different means to structure application code and manage control flow. The event loop sequentially processes queued events by executing the associated callback of the event. Callbacks are functions that have been registered earlier as the event handler for certain types of events. Callbacks are assumed to be short-running functions that do not block the CPU for a longer period, since this would block the entire event loop. Instead, callbacks are usually short functions that might dispatch background operations, that eventually yield new events. Support for anonymous functions and closures are an important feature of programming languages for event-driven programming, because the first-class functions are used to define callbacks, and closures can provide a substitute for context and continuation. Due to the fact that closures bind state to a callback function, the state is preserved in the closure and is available again once the callback is executed in the event loop. The notion of single-threaded execution and callbacks also provides implicit coordination, since callbacks are never executed in parallel and each emitted event is eventually handled using a callback. It is safe to assume that no other callback runs in parallel, and event handlers can yield control and thus support cooperative scheduling. Once a callback execution has finished, the event loop dequeues the next event and applies its callback. Event-driven frameworks and platforms provide an implicit or explicit event loop, an asynchronous API for I/O operations and

1 http://nodejs.org/
2 http://twistedmatrix.com/
3 http://rubyeventmachine.com/
4 http://poe.perl.org/

system functions, and means to register callbacks to events.

Using a single thread renders the occurrence of a deadlock impossible. But when developers do not fully understand the implications of an event loop, issues similar to starvations and race conditions can still appear. As long as a callback executes, it consumes the only thread of the application. When a callback blocks, either by running blocking operations or by not returning (e.g. infinite loops), the entire application is blocked. When multiple asynchronous operations are dispatched, the developer must not make any assumptions on the order of execution of their callbacks. Even worse, the executions of the callbacks can interleave with other callbacks from other events. Assumptions on callback execution ordering are especially fatal when global variables instead of closures are used for holding state between callbacks.

5.5.3 Case Study: Concurrency in node.js

Node.js is a platform built on the v8 JavaScript engine of Google's Chrome browser, an event library for asynchronous, non-blocking operations and some C/C++ glue code [Til10]. It provides a lightweight environment for event-driven programming in JavaScript, using a non-blocking I/O model. Compared to other event-driven frameworks and platforms, node.js sticks out due to several reasons. Unlike many other programming languages, JavaScript has neither built-in mechanisms for I/O, nor for concurrency. This allows to expose a purely asynchronous, non-blocking API for all operations and prevents the accidental use of blocking calls (in fact, there are also blocking API calls for special purposes). With its good language support for anonymous functions, closures and event handling, JavaScript also supplies a strong foundation of primitives for event-driven programming. The event loop is backed by a single thread, so that no synchronization is required at all. Merging multiple event sources such as timers and I/O notifications and sequentially queueing events is also hidden inside the platform.

Node.js does not expose the event loop to the developer. A node.js application consists of a sequence of operations that might provide asynchronous behavior. For each asynchronous operation, a proper callback should be specified. A callback, in turn, can dispatch further asynchronous operations. For instance, a node.js web server uses a single asynchronous function to register an HTTP server, passing a callback. The callback defines function to be executed each time a new request is received. The request handling callback in turn might execute file I/O, which is also asynchronous. This results in heavy callback chaining and an obvious inversion of control due to the callback-centric style. This outcome requires strict coding rules or the assistance of libraries in order to keep code readable and manageable without losing track of the flow of execution. One of these libraries is async[1], which makes heavy use of functional properties of JavaScript. It provides support for control flow patterns such as waterfall, serial and parallel execution, functional concepts like map, reduce, filter, some and every, and concepts for error handling. All these abstractions make callback handling more reasonable. Although node.js is

1 `https://github.com/caolan/async`

a rather new platform, there are already many libraries and extensions available, especially for web application development. One of the most distinguished libraries for node.js is socket.io[1]. It provides web realtime communication mechanisms between a server and browsers, but abstracts from the underlying transport. Hence, the developer can use send functions and message event handlers both for server-side and client-side application code. The underlying communication uses WebSockets, but supports several fallbacks such as long polling, when not available.

Listing 5.5 illustrates, how shared application state is handled in a single-thread event-driven application, in this case node.js. The application defines a global `count` variable. Then, a server is created with a callback for request handling. The callback responds to every request by incrementing the `count` variable and returning its value. Thanks to the single-threaded execution model, there is no synchronization required when accessing or modifying the variable.

5.5.4 Event-driven Concurrent Application Logic

It is obvious that single-threaded, event-driven application servers cannot reduce latencies of CPU-bound requests. Most platform components are part of our distributed architecture, hence most of the operations of a request are in fact I/O-bound. For complex computations, we also provide a decoupled background worker pool. Thus, the event-driven approach is a suitable concept when the application logic primarily dispatches work tasks, calls, and requests to the platform components and later combines the results without high computational efforts. The advantages of the event-driven approach for I/O-bound parallelism have already been illustrated in chapter 4. When shared application state and pub/sub coordination mechanisms are outsourced to platform components, this mechanism fits well into the event-driven approach. Actually, if application state can be isolated to separate application servers (e.g. sessions of interactive web applications), another benefit of the event-driven approach comes even more handy. The single-threaded execution models makes it very easy to access and modify mutable state between requests, because no real concurrency is involved.

```
var http = require('http');

var count = 0;

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end((++count)+'\n');
}).listen(8080);
```

Listing 5.5: A minimalistic, concurrent web application written in JavaScript that returns the number of requests handled so far, using the node.js platform.

1 http://socket.io/

5.6 Other Approaches and Concurrency Primitives

We have now seen four common concurrency concepts that can be used for programming concurrent application logic. However, there are still other concurrency concepts and language primitives for concurrent programming. We will now introduce the more interesting ones briefly. We give hints how they relate to the former concepts and how they can be used in practice.

5.6.1 Futures, Promises and Asynchronous Tasks

Promises [Fri76, Lis88] and futures [Bak77] generally describe the same idea of decoupling a computation and its eventual result by providing a proxy entity that returns the result, once available. In concurrent programming, promises and futures are often used for asynchronous background tasks. Once a task is dispatched, the proxy entity is returned and the caller's flow of execution can continue, decoupled from the new computation. The proxy entity can later be queried to return the result of the decoupled computation. If the result is not yet available, the proxy entity either blocks or provides a notification, when non-blocking.

Promises and futures also introduce a synchronization mechanism, as they allow to dispatch independent computations, but synchronize with the initial flow of control, once the result is requested and eventually returned. Futures and promises are available in many programming languages using threads for concurrency. In this case, the execution of the background task is scheduled to another thread, allowing the initial thread to continue its execution. Actor-based systems often use an actor as a proxy entity of a computation, which is essentially the same as a future [Hew73]. Sending a message to another actor and awaiting its eventual response is also often abstracted using futures. In event-driven systems, eventual results are represented by events, and handled by associated callbacks.

When programming web applications, promises or futures can be used to decrease latency. The scatter-gather pattern dispatches multiple independent operations (scatter) and waits for all operations to yield results (gather). For instance, multiple independent database queries can be parallelized using this pattern. This can be implemented by scattering all operations as background tasks yielding proxy entities, then gathering all results by accessing the proxy entities. In effect, this converts a sequence of operations into parallel execution of operations. Dataflow programming provides a similar execution strategy, but hides the notion of futures in the implementation.

5.6.2 Coroutines, Fibers and Green Threads

Coroutines [Con63], and similarly fibers and green threads, are a generalization of subroutines. While a subroutine is executed sequentially and straightly, a coroutine can suspend and resume its execution at distinct points in code. Thus, coroutines are good primitives for cooperative task handling, as coroutines are able to yield execution. We have identified the advantages of

cooperative task handling in chapter 4, especially in case of massive parallelism of asynchronous operations such as I/O.

Coroutines are often used as low-level primitives, namely as an alternative to threads for implementing high-level concurrency concepts. For example, actor-based systems and event-driven platforms may use coroutines for their underlying implementation.

There are also several programming languages and language extensions that introduce coroutines or their variants to high-level programming languages. For instance, greenlet[1] is a coroutine framework for Python, that is heavily used by high performance event loop frameworks such as gevent[2].

Google Go[3] is a general-purpose programming language from Google that supports garbage collection and synchronous message passing for concurrency (see below). Go targets usage scenarios similar to C/C++ by tendency. It does not supply threads for concurrent flows of executions, but a primitive called goroutine, which is derived from coroutines. Goroutines are functions that are executed in parallel with their caller and other running goroutines. The runtime system maps goroutines to a number of underlying threads, which might lead to truely parallel execution. In other circumstances, multiple goroutines might also be executed by a single thread using cooperative scheduling. Hence, they are more powerful than conventional coroutines, as they imply parallel execution and communicate via synchronous message passing, and not just by yielding.

5.6.3 Channels and Synchronous Message Passing

Message passing is a theoretical model for concurrent systems that became well-known thanks to Hoare's CSP [Hoa78]. It is also the theoretical foundation for concurrent programming concepts.

There are two different flavors of message passing—synchronous and asynchronous. We have already got to know the latter one, because the actor model is essentially built on asynchronous message passing between actors. Asynchronous message passing decouples communication between entities and allows senders to send messages without waiting for their receivers. In particular, there is no synchronization necessary between senders and receivers for message exchange and both entities can run independently. On the other hand, the sender can not know when a message is actually received and handled by the recipient.

The other variant, synchronous message passing, uses explicit message exchanging. Both the sender and receiver have to be ready and block while the message is getting exchanged. As a consequence, synchronous message passing yields a form of synchronization, because message exchanges are explicit synchronization points for different flows of control.

There are several other differences between both models. In asynchronous message passing

1 http://codespeak.net/py/0.9.2/greenlet.html
2 http://www.gevent.org/
3 http://golang.org/

models, the communicating entities have identities, while their synchronous counterparts are anonymous. Synchronous messaging uses explicit, named channels for communication between entities, while asynchronous messaging does not have intermediaries.

Google Go makes heavy use of synchronous message passing for concurrent programming, very similar to the way pipes are used in Unix (also synchronous). Channels are first-level languages primitives that are used for data exchange between goroutines. As goroutines can be scheduled to multiple threads and might actually run in parallel, channels are also the main synchronization primitive assuring that multiple goroutines meet in a mutually known state.

5.6.4 Dataflow Programming

Dataflow programming with declarative concurrency is a very elegant, yet rather uncommon approach to tackle concurrency. Imperative programming is based on the idea of describing a sequence of explicit operations to execute. Instead, dataflow programming defines the relations between necessary operations, yielding an implicit graph of dependencies, the flows of execution. This allows the runtime systems to automatically identify independent steps of operations and parallelize them at runtime. Coordination and synchronization are hidden in the runtime system, often by using channels in order to wait for multiple input data and then initiate the next steps of computation. Although this programming concept allows true parallelism, coordination and synchronization efforts are hidden due to the declarative style.

Mozart/Oz [Roy04] is a programming language that supports multiple paradigms and has strong support for dataflow programming with declarative concurrency. GPars[1] is a Java concurrency library that also supplies a rich set of dataflow concurrency primitives.

Dataflow programming is also interesting for web application development. Application-level request handling can be defined as a flow of operations. Thanks to the implicit parallelization, independent operations are then executed in parallel and later synchronized without any locking. Hence, this style helps to decrease latencies.

5.7 Summary

Conventional web applications are primarily "embarrassingly parallel", because requests are entirely isolated and easy to execute independently. Highly interactive and collaborative web applications, as well as real-time web applications, require more challenging concurrency properties. On the one hand, these applications need to coordinate actions and communicate between different concurrent requests. They also share a common state between requests as part of the application logic. This enables features such as instant notifications, asynchronous messaging and server-sent events—important building blocks for a wide range of modern web applications

1 `http://gpars.codehaus.org/`

including social web applications, multiplayer browser games, collaborative web platforms and web services for mobile applications. On the other hand, modern applications require increasingly low latencies and very high responsiveness. This can be tackled by parallelizing the application logic operations which are necessary for a request as much as possible. In turn, this requires synchronization of the request logic for each request. Hence, the application logic of modern, large-scale web applications is inherently concurrent. We have therefore presented and analyzed the most popular concurrency concepts for programming concurrent systems in general.

The concept of threads and locks is based on mutable state shared by multiple flows of execution. Locks are required to enforce mutual exclusion and prevent race conditions. In turn, locks introduce the hazard of deadlocks, livelocks and starvation. Choosing the right locking granularity becomes a trade-off between the danger of race conditions and deadlocks, and between degenerated sequential execution and unmanageable nondeterminism. The combination of nondeterminism and mutable shared state makes it extremely hard to reason about lock-based concurrency and its correctness. Furthermore, compositions of lock-based components are not guaranteed to be deadlock-free. Concurrent programming based on threads, locks and shared state is still essential. It represents the low-level foundation for all higher concurrency abstractions and cannot be ignored. However, it is often the wrong level of abstraction for high-level application programming because it is too difficult and too error-prone.

STM addresses these problems of lock-based concurrency by introducing transactional operations for concurrent code. Transactions isolate operations on mutable shared state and are lock-free and composable. However, transactional code must not contain any side effects (e.g. no I/O operations), as it might be aborted and retried. STM hides the complexity of transaction handling in a software layer, which introduces some computational overhead at runtime.

The actor model represents an entirely different approach that isolates mutability of state. Actors are separate, single-threaded entities that communicate via immutable, asynchronous and guaranteed messaging. Thus, actors encapsulate state and provide a programming model that embraces message-based, distributed computing.

Common event-driven architectures evict the danger of deadlocks by using a single-threaded event loop. The event loop dequeues piled-up events and sequentially executes each event with its associated event handler (i.e. callbacks). Hence, application logic is defined as a set of event handlers, which results in an inversion of control when compared to purely sequential code. When applications are primarily I/O-bound and event handlers do not execute computationally complex operations, this concept utilizes a single CPU core very well and makes concurrency reasoning very easy.

We have also seen that dataflow programming uses declarative concurrency based on implicit dependency graphs. Synchronous message passing provides channel-based points for data exchange, which can also be used for synchronizing and coordinating different flows of execution. Coroutines represent primitives for cooperative task handling. Futures and promises are helpful for decoupling background tasks and later synchronizing with their eventual result.

6 Concurrent and Scalable Storage Backends

In the previous chatpers, we have always to evade state as much as possible. Our considerdations for web server internals are based on a shared-nothing architecture. For programming application logic, we have given the preference to reduce the usage of shared, mutuable state to the bare minimum. However, only a handful of web applications can completely disregard any kind of state. Statelessness is viable for validation or computation services, and a few other applications. For virtually all other web applications, some type of persistent state represents an essential part of the overall application. In our architectural model, we have edged away state handling to dedicated backend services.

In this chapter, we consider the impact of concurrency and scalability to storage backends. We illustrate the challenge of guaranteeing consistency in distributed database systems and point to different consistency models. We then outline some internal concepts of distributed database systems and describe how they handle replication and partitioning. Finally, we introduce differnt types of distributed database systems and assess their features.

6.1 The Challenge of Distributed Database Systems

As we think about large-scale web applications, we need storage backends that scale and support concurrency. By scalability, we aim for increasable data capacity and growing read/write throughput of a high degree. The application servers in our model handle huge numbers of requests in parallel. As they, in turn, rely on the backend storage systems, we need to cope with highly concurrent access at this level as well.

Throughput and capacity increases can only be sustainably achieved by employing a horizontal scale mechanism. A single database server would only be able to scale up to a certain load, even with specialized hardware equipment. As a consequence, we need a concurrent and distributed system for scalable backend storage.

The backend storage must persistently hold the application state, thus we are also expecting some kind of data consistency when reading/writing data in our application. We are dealing

with a distributed system, so we must expect failures in advance. In case of a failure of a single node, we still want the overall storage system to operate seamlessly and maintain its availability. Likewise, we are executing storage operations as part of web requests, thus we demand low latency of operations.

These general requirements lead us to an important barrier of distributed systems, Brewer's theorem on the correlation of consistency, availability and partition tolerance.

6.1.1 The CAP Theorem

The CAP conjecture was introduced by Brewer in 2000 [Bre00] and later confirmed by Gilbert and Lynch [Gil02] as a theorem. Brewer argues that distributing computations is relatively easy, but the hard part of distributed systems is actually persisting state. Moreover, he postulates three distinct properties for distributed systems with an inherent correlation.

Consistency The consistency property describes a consistent view of data on all nodes of the distributed system. That is, the system assures that operations have an atomic characteristic and changes are disseminated simultaneously to all nodes, yielding the same results.

Availability This property demands the system to eventually answer every request, even in case of failures. This must be true for both read and write operations. In practice, this property is often narrowed down to bounded responses in reasonable time. However, Gilbert and Lynch have confirmed the theorem even for unbounded, eventual responses.

Partition tolerance This property describes the fact that the system is resilient to message losses between nodes. A partition is an arbitrary split between nodes of a system, resulting in complete message loss in between. This affects the prior properties. Mechanisms for maintaining consistency must cope with messages losses. And according to the availability property, every node of any potential partition must be able to respond to a request.

The core statement of Brewer's theorem now goes as follows:

> "You can have at most two of these properties for any shared-data system."

We have seen that all properties are desirable. But any real system must trade off the properties and dismiss at least one of them, as shown in figure 6.1. So we have three distinct combinations with significantly different characteristics.

Consistency & Availability (CA)

The group of distributed systems following this model provides a consistent and available service, but does not tolerate partitions. In case of a partition, these systems may become inconsistent, as we will see soon. The combination is also known as *high-availability consistency*.

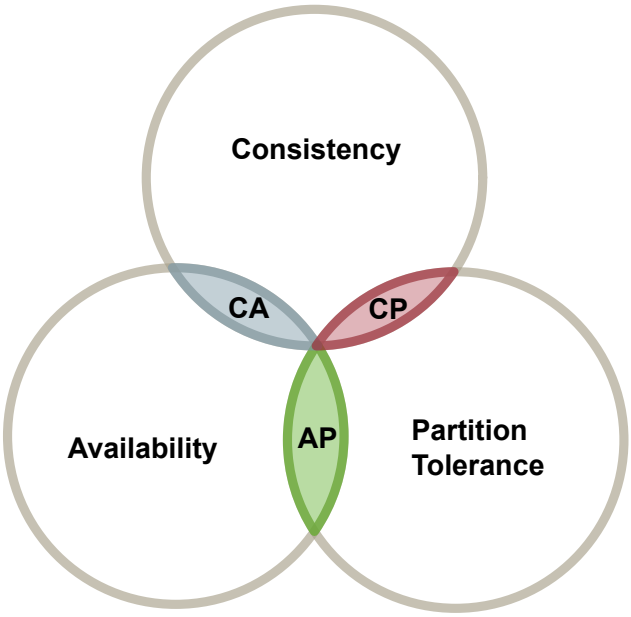

Figure 6.1: Different properties that a distributed system can guarantee at the same time, according to Brewer's theorem [Breoo].

Contenders following this approach include most of the traditional relational database management systems. Replication is an important mechanism for achieving highly available consistency and transaction protocols such as the two-phase commit protocol are applied to ensure consistency. The separation into partitions may lead to so-called "split brain" scenarios, in which different partitions create conflicting replicas as a result of isolation.

Recovering from such scenarios would require some kind of consensus protocol. This in turn would disallow nodes to service requests unless a consensus is available. We would thus convert our CA approach into a CP approach at the sacrifice of availability. The shortcomings of coping with network errors renders the CA approach less suitable for larger distributed database systems.

Consistency & Partition Tolerance (CP)

Distributed systems at the intersection of consistency and partition tolerance provide a strongly consistent service. Consistency is guaranteed even in the presence of a partition. However, nodes of a partition may not be able to respond to requests as long as they have not yet agreed with other nodes that may be temporarily unreachable. As a result, availability cannot be always provided. This combination of properties is also known as *enforced consistency*.

In practice, this approach is important when consistency must be enforced at any costs, and the system is still inherently distributed by design. That is for instance a banking application, where the balance of all accounts is a primary constraint. Database systems implementing this

model are also often based on relational database systems. Supporting consistent states even in case of network errors requires the usage of sophisticated algorithms for quorum and majority decisions. Such a protocol for solving consensus is the Paxos protocol [Lam98].

Availability & Partition Tolerance (AP)

Distributed systems that are always available and tolerate partitions maintain their service even when partitions occur. However, a reachable node may then hold (temporarily) inconsistent state. Thus, this intersection of properties is also known as *eventual consistency*.

In terms of data and state, the sacrifice of strong consistency guarantees appears to be questionable at first glance. However, many applications can indeed tolerate deferred consistency properties when favoring availability at all costs. In this case, it is important to keep in mind potential issues due to eventually consistent data on application level during development. Well known examples following this approach are the Domain Name System (DNS) or web caches. Stale data (e.g. host mappings resp. cached responses) are acceptable for a while, but eventually the latest version of the data disseminates and purges older entries.

Criticism and an Alternative Model to the CAP Theorem

With the beginning of the NoSQL movement and increasing interest in eventually consistent data stores, some criticism has been leveled at the CAP theorem. A central issue of the CAP theorem results from the simplifying error model that only targets network failures. It is especially the premature dropping of consistency as the answer to network errors, that is raised to question by members of the database community such as Stonebraker[Sto10].

Abadi targets other shortcomings of the CAP theorem, namely the asymmetry of availability and consistency and the generalising trade-off between consistency and availability [Aba10]. According to Abadi, this becomes obvious when regarding systems in the absence of partitions. As a result, Abadi proposes an alternative model: PACELC. It can be used to describe whether a system, in case of a partition (P), either focuses availability (A) or consistency (C), and whether a system else (E) focuses on latency (L) or consistency (C). As a consequence, systems can now be categorized more precisely. As an example, eventually consistent systems (AP in terms of CAP) can be split up into PA/EL or PA/CL systems, yielding more details on their regular operational mode in the absence of partitions.

6.1.2 Consistency Models

We have seen that consistency, availability and partition tolerance cannot be guaranteed at the same time for a distributed system. From our perspective of large-scale web architectures, this is mainly important for the storage backends, as these components keep our application states persistently. When building a large web architecture, we have to choose storage components while keeping in mind the prior limitations. Consistency is the most defining constraint for our

application logic, as it determines the type of consistency to expect from the database system. Thus, we need to consider different consistency models and their impacts on the application. We limit our review to the two important main consistency models, from the perspective of our application, which is the client of the database.

While the following two models are generally opponents of each other, it is noteworthy that Brewer mentions they in fact form a spectrum. As a result, there are some means for trade-offs in between. Several distributed database systems allow to fine-tune consistency trade-offs via configuration parameters. Also, a large-scale web application can often split data storage requirements into functional groups. For instance, an e-commerce site can tolerate relaxed consistency guarantees for product ratings and comments, while stronger consistency is required for goods in stock and orders.

ACID

Relational Database Management Systems (RDBMSs) are the predominant database systems currently in use for most applications, including web applications. Their data model and their internals are strongly connected to transactional behaviour when operating on data. However, transactional behaviour is not solely related to RDBMSs, but is also used for other systems. A set of properties describes the guarantees that database transactions generally provide in order to maintain the validity of data [Hae83].

***A**tomicity* This property determines that a transaction executes with a "all or nothing" manner. A transaction can either be a single operation, or a sequence of operations resp. sub-transactions. As a result, a transaction either succeeds and the database state is changed, or the database remains unchanged, and all operations are dismissed.

***C**onsistency* In context of transactions, we define consistency as the transition from one valid state to another, never leaving behind an invalid state when executing transactions.

***I**solation* The concept of isolation guarantees that no transaction sees premature operations of other running transactions. In essence, this prevents conflicts between concurrent transactions.

***D**urability* The durability property assures persistence of executed transactions. Once a transaction has committed, the outcome of the transaction such as state changes are kept, even in case of a crash or other failures.

Strongly adhering to the principles of ACID results in an execution order that has the same effect as a purely serial execution. In other words, there is always a serially equivalent order of transactions that represents the exact same state [Dol05]. It is obvious that ensuring a serializable order negatively affects performance and concurrency, even when a single machine is used. In fact, some of the properties are often relaxed to a certain extent in order to improve performance. A

weaker isolation level between transactions is the most used mechanism to speed up transactions and their throughput. Stepwise, a transactional system can leave *serializablity* and fall back to the weaker isolation levels *repeatable reads*, *read committed* and *read uncommitted*. These levels gradually remove range locks, read locks and resp. write locks (in that order). As a result, concurrent transactions are less isolated and can see partial results of other transactions, yielding so called read phenomena. Some implementations also weaken the durability property by not guaranteeing to write directly to disk. Instead, committed states are buffered in memory and eventually flushed to disk. This heavily decreases latencies at the cost of data integrity.

Consistency is still a core property of the ACID model, that cannot be relaxed easily. The mutual dependencies of the properties make it impossible to remove a single property without affecting the others. Referring back to the CAP theorem, we have seen the trade-off between consistency and availability regarding distributed database systems that must tolerate partitions. In case we choose a database system that follows the ACID paradigm, we cannot guarantee high availability anymore. The usage of ACID as part of a distributed systems yields the need of distributed transactions or similar mechanisms for preserving the transactional properties when state is shared and sharded onto multiple nodes.

Now let us reconsider what would happen if we evict the burden of distributed transactions. As we are talking about distributed systems, we have no global shared state by default. The only knowledge we have is a per-node knowledge of its own past. Having no global time, no global now, we cannot inherently have atomic operations on system level, as operations occur at different times on different machines. This softens isolation and we must leave the notion of global state for now. Having no immediate, global state of the system in turn endangers durability.

In conclusion, building distributed systems adhering to the ACID paradigm is a demanding challenge. It requires complex coordination and synchronization efforts between all involved nodes, and generates considerable communication overhead within the entire system. It is not for nothing that distributed transactions are feared by many architects [Helo9, Alv11]. Although it is possible to build such systems, some of the original motivations for using a distributed database system have been mitigated on this path. Isolation and serializablity contradict scalability and concurrency. Therefore, we will now consider an alternative model for consistency that sacrifices consistency for other properties that are interesting for certain systems.

BASE

This alternative consistency model is basically the subsumption of properties resulting from a system that provides availability and partition tolerance, but no strong consistency [Prio8]. While a strong consistency model as provided by ACID implies that all subsequent reads after a write yield the new, updated state for all clients and on all nodes of the system, this is weakened for BASE. Instead, the weak consistency of BASE comes up with an inconsistency window, a period in which the propagation oft the update is not yet guaranteed.

***B**asically **a**vailable* The availability of the system even in case of failures represents a strong feature of the BASE model.

***S**oft state* No strong consistency is provided and clients must accept stale state under certain circumstances.

***E**ventually consistent* Consistency is provided in a "best effort" manner, yielding a consistent state as soon as possible.

The optimistic behaviour of BASE represents a best effort approach towards consistency, but is also said to be simpler and faster. Availability and scaling capacities are primary objectives at the expense of consistency. This has an impact on the application logic, as the developer must be aware of the possibility of stale data. On the other hand, favoring availability over consistency has also benefits for the application logic in some scenarios. For instance, a partial system failure of an ACID system might reject write operations, forcing the application to handle the data to be written somehow. A BASE system in turn might always accept writes, even in case of network failures, but they might not be visible for other nodes immediately. The applicability of relaxed consistency models depends very much on the application requirements. Strict constraints of balanced accounts for a banking application do not fit eventual consistency naturally. But many web applications built around social objects and user interactions can actually accept slightly stale data. When the inconsistency window is on average smaller than the time between request/response cycles of user interactions, a user might not even realize any kind of inconsistency at all.

For application development, there are several more specific variants of eventual consistency that give the developers certain types of promises and allow to reason more easily about relaxed consistency inside the application logic [Vog08]. When *causal consistency* is provided, operations that might have a causal relation are seen in the same sequence by all nodes. *Read-your-writes consistency* ensures that a client will always see the new value after having updated it, and never again an older value. *Session consistency* is a variant of the read-your-writes consistency, that limits the behavior to a certain client session. It hence requires session affinity inside the system, but can be helpful for developers. The guarantee of *monotonic read consistency* is that after a read operation, any subsequent read must not yield older values. Similarly, *monotonic write consistency* guarantees to serialize write operations of a single client. Not providing this guarantee represents a severe issue for application developers, making it very difficult to reason about states.

Some of these consistencies can be combined or are subset of others. Providing both monotonic read and read-your-writes consistency provides a reasonable foundation for developing applications based on eventual consistency.

As eventual consistency entails the possibility of data conflicts, appropriate resolution mechanisms must be employed. Often, conflict resolution is also part of the application logic, and not only provided by the database system. In general, conflicts are resolved either on read or write operations, or asynchronously in the background, decoupled from client operations.

6.2 Internals of Distributed Database Systems

Developing distributed database systems is not a simple task, and it requires concepts from both the database community and the distributed systems community. In our brief overview, we examine additional building blocks that are necessary when designing databases with distribution support. We also list some replication and partitioning strategies for distributed database systems.

6.2.1 Building Blocks

Hellerstein and Stonebraker provide a very comprehensible introduction to traditional database internals [Helo7]. Such internals include indexes and data structures, I/O handling components, transaction handling, concurrency control, query processing and client communication interfaces. As these concepts are common for database systems, we focus on some of the necessary building blocks for distributed database systems instead, including transactiobn management, concurrency control, data versioning, interfaces, and scalable data partitioning and parallel data processing.

Distributed Transaction Management

In a non-distributed scenario, handling concurrent transactions is generally easier, because everything happens locally on a single machine. Distributed transactions handle operations with transactional behavior between multiple nodes. Thus, a transaction in a distributed system must either be applied to all participating nodes, or to no one at all. Distributed transactions are more difficult to implement due to the risk of network errors, (partial) failures of nodes and non-locality. A basic component for distributed transactions is a coordinating service that manages and coordinates transactions between all participants, based on a transaction protocol.

Popular protocols are the Two-phase Commit Protocol (2PC) [Lam79a] and the Three-phase Commit Protocol (3PC) [Ske83]. 2PC separates a voting phase and a completion phase, but it is blocking and not fault-tolerant. 3PC addresses the drawbacks of 2PC by additional coordination steps. However, 3PC cannot cope with network partitions.

Alternatively, quorum-based voting protocols can be used for committing transactions in distributed setups [Ske82]. The underlying idea is to mark a transaction as executed, when the majority of nodes have executed it. So either the abort quorum or the commit quorum must be obtained for termination. The Paxos [Lam98] protocol family provides consensus solving that can be used for quorum-based voting.

Concurrency Control and Data Versioning

The inherent parallelism states a problem for distributed database systems, especially when concurrent write operations are allowed on different nodes. In particular, relaxed consistency guarantees and the acceptance of network partitions require concepts for data versioning and controlling concurrent operations on data.

Distributed concurrency control mechanisms can be generally divided into pessimistic and optimistic algorithms and—ortoghonally—into locking-based, timestamp-based or hybrid algorithms. Pessimistic algorithms provide conflict prevention by strict coordination of concurrent transactions, while optimistic algorithms do not expect regular conflicts and delay conflict checking to the end of a transaction life-cycle. Locking-based algorithms use explicit locks for operations in order to prevent conflicts. Popular locking-based algorithms include traditional two-phase-locking [Esw76] and its variants. Also quorum-based voting can be applied, using read and write quorums for the corresponding operations [Gif79].

Distributed database systems also take advantage of timestamp-based concurrency control mechanisms, such as MVCC [Ber81, Ree78]. We have already encountered MVCC as an underlying implementation for STM systems in chapter 5. Timestamp-based mechanisms use logical clocks to identify either data changes or transactions over time. The logical ordering allows to reason about the sequence of operations and to protect from conflicts. Several distributed database systems use vector clocks [Lam78] for versioning data entries in face of a network partition [DeC07]. The version history then allows to reason about conflicts and facilitate merges.

Data Interfaces and APIs

Traditional database systems are sometimes entirely embedded into an application, or they use arbitrary protocols for access. Software components such as Java Database Connectivity API (JDBC) or Open Database Connectivity (ODBC) abstract from concrete database protocols and supply generic APIs. For distributed database systems, interfaces for remote access are obligatory. Hence, established distributed technologies for communication and data serialization are often integrated from the beginning. These technologies facilitate application integration and testing.

Database calls and queries are often dispatched using RPC invocations or HTTP requests. The RPC approach uses framework bindings like Thrift [Aga07]. Data is interchanged using serialization technologies such as Thrift's own serialization[1] or Google's Protocol Buffers[2]. HTTP-based APIs often emulate some of the REST principles. For serialization, formats like JSON, BSON[3] (a binary JSON representation) or XML are then used. While low-level RPC calls generally provide a slightly better performance due to less overhead, the usage of HTTP-based APIs introduces HTTP concepts like caching for free.

1 http://thrift.apache.org/
2 http://code.google.com/p/protobuf/
3 http://bsonspec.org/

Scalable Data Partitioning

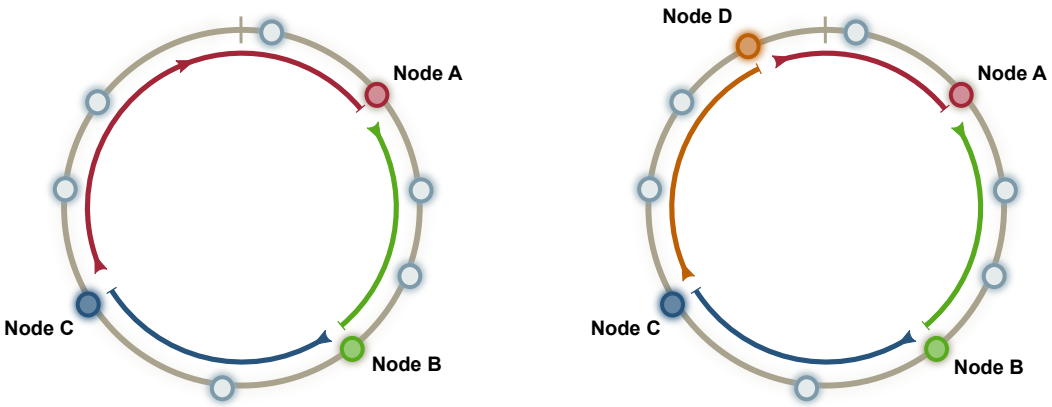

Figure 6.2: Consistent hashing maps nodes and data items into the same ring for partitioning. The left illustration shows a set of data items (gray) mapped to three nodes. On the right side, the additional Node D has joined the system and is also mapped into the ring. As a consequence, only a small sector of the ring is affected from repartitioning. Node D takes over two data items that have formerly been assigned to Node A.

Allocating large amounts of data to a number of nodes becomes more complex, if data scalability is required and the number of available nodes changes. Scaling out means supplying additional nodes, often at runtime in the first place. Sometimes, also scaling back to less nodes is interesting, when the amount of data decreases. Appropriate strategies are required, how to partition and how to allocate data when scaling in and out.

Traditional setups with a fixed number of hosts often allocate data by applying a hash function on a data item (e.g. the key), then using the result *modulo* the number of nodes in order to calculate the node responsible for the item. The strategy is straightforward, but it fails when the number of nodes changes. Recalculating and redistributing all items due to changed partitioning keys is then necessary, but not reasonable in practice. One way to approach this problem is consistent hashing [Kar97]. The fundamental idea of consistent hashing is to hash data items and nodes into a common ring using the same hash function. The algorithm determines that each data item has to be stored by the next clockwise adjacent node in the ring, as shown in figure 6.2. When new nodes are added to the ring, or nodes leave the ring, a small sector of the ring is affected and only the data items in this sector must be reallocated. In essence, consistent hashing is a partitioning strategy that works with varying number of nodes and provides a consistent mapping that prevents an unnecessary reallocating of data when the amount of nodes scales.

Parallel Data Processing

Processing data entries in a distributed database system is necessary for several operations. For instance, generating indexes requires the execution of the same operations on all data entries and machines. In several non-relational database systems, it is the developer's task to implement index generating, hence appropriate programming models are required for such *embarrassingly parallel* tasks.

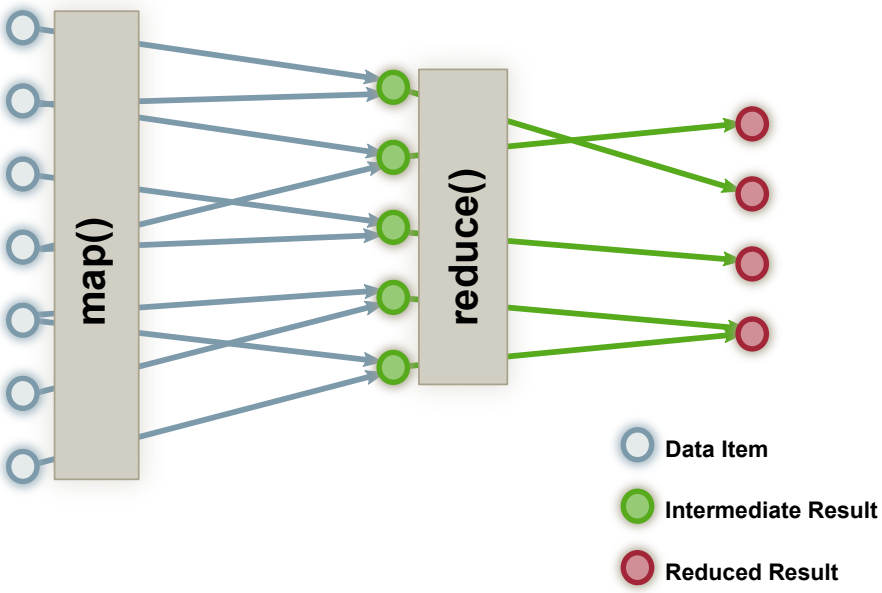

Figure 6.3: A schematic illustration of the phases of a MapReduce computation. The `map()` function is applied to all items and produces intermediate results. Grouped by their key, these intermediate results get merged using the `reduce()` function.

A popular approach is the MapReduce model [Dea08], which is inspired by functional programming languages. It separates parallel processing of possibly large data sets into two steps, as shown in figure 6.3. The `map` function takes data entries and emits intermediate key-value pairs. Next, all intermediate pairs are grouped by keys. The `reduce` function is then applied to all intermediate pairs with the same key, yielding simple values as a result. Distribution, coordination and execution is managed by the framework resp. database system, so the developer only has to provide the map and reduce function. This principle easily allows tasks such as counting or sorting on large data sets. MapReduce is also used for building indexes, either using the sorted intermediate key-value pairs, or using the sorted reduced results.

6.2.2 Replication and Partitioning Strategies

Replication and partitioning are necessary concepts for distributed database systems. Replication is responsible for data distribution between nodes. On the basis of replication, availability can be increased and fail-over mechanisms can be deployed for fault-tolerance. Replication can also help to scale read operations. Partitioning deals with the challenge of scaling out large amounts of data.

Replication

There are various forms of replication for distributed systems, but not all are applicable for database systems that target availability and highly concurrent access. Replication mechanisms can be either synchronous or asynchronous, active or passive, and they have different propagation characteristics [Dolo5, Moi11].

Synchronous replication provides atomic semantics for operations, that are backed by all running replicas. This requires distributed transaction protocols for coordination. *Asynchronous replication* allows a single node to acknowledge an operation independently. Other nodes will eventually receive the updates. But, as opposed to synchronous replication, immediate updates of all nodes are not guaranteed. Asynchronous replication works either periodically or aperiodically.

In *active replication*, all nodes receive and process a request (e.g. write operation), and coordinate their response. *Passive replication* favors a designated primary that processes the request and updates the other nodes afterwards. In case of a fail-over, a secondary takes over the service.

The propagation aspects determine, how read and write operations from clients are handled, and how updates disseminate to the replicas. In a *master-slave* setup, writes are processed by a single master. As web applications tend to issue more read requests than write requests, many setups take advantage of this inherent property and provide a single master server and multiple slaves [Sch08]. The master server is solely issued for write requests, and all read requests are load-balanced to one of the slaves. Obviously, this setup does only help to scale read operations, but not write operations. A *multi-master* setup allows multiple nodes to accept writes. This indeed increases write scalability. However, it requires conflict management strategies, as simultaneous writes on the same data may lead to inconsistencies. *Quorum-based* systems [Gif79] allow to fine tune, how many replicas must be accessed for reading operations, how many replicas store the data, and how many replicas must acknowledge update operations. These three parameters directly affect the strengths of consistency and fault-tolerance. In figure 6.4, we can see two exemplary replication strategies in use for web applications.

Common replication strategies include snapshot replication, transactional replication, merge replication and statement-based replication [Moi11]. Snapshot replication is based on periodic copying of all data entries. These snapshots are then forwarded and applied on replicas. Transactional replication employs a transactional behavior for changes using distributed transactions between servers. Merge replication allows for partition tolerance and offline nodes, since it

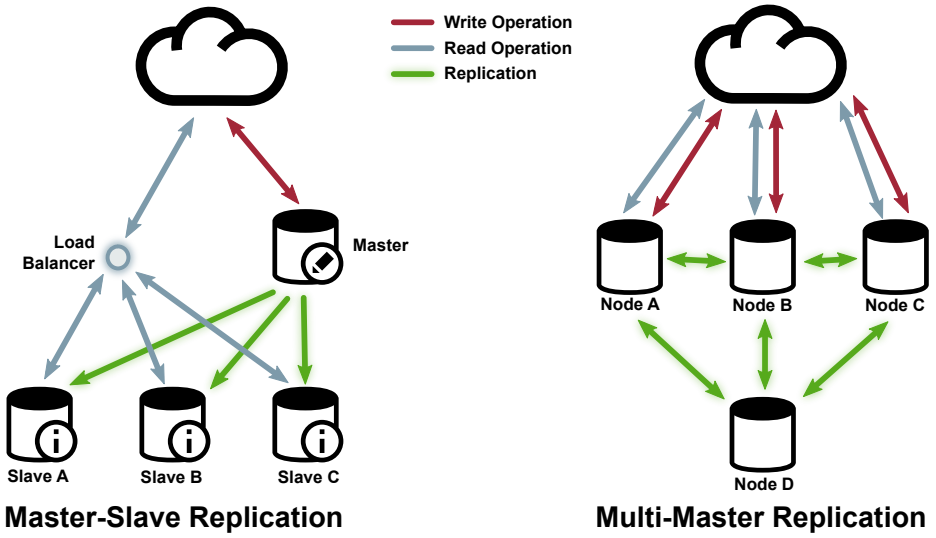

Figure 6.4: Two replication setups for the backend of web application. On the left side, we can see a master-slave setup that is often used by MySQL. The master handles writes and asynchronously updates the slaves. Read requests are load-balanced to the slaves. On the right side, a common replication setup for CouchDB is depicted. Multiple masters handle all requests and perform asynchronous merge replication that might require conflict resolution.

synchronizes data when nodes eventually become available. Conflict resolution strategies are necessary to handle conflicting changes. Statement-based replication forwards database queries to replicas. Read queries can be forwarded to a single instance, while queries including write operations are forwarded to all instances.

Partitioning

There are different partitioning approaches for scaling out large amounts of data: functional, vertically and horizontal [Allo8, Scho8]. *Functional partitioning* separates distinct parts of the application data that are not dependent on each other. For instance, customer data, product data and inventory data are not just stored in different tables, but can also be stored on different instances of database nodes. *Vertical partitioning* targets data partitioning, that is the efficient allocation of a complex data row into tables. Normalization and denormalization are typical mechanisms for vertical partitioning. For instance, "row splitting" separates a single table into multiple tables, thereby separating columns that change often from columns that are rather static. Such a split can also improve performance. *Horizontal partitioning*, also known as sharding, addresses the problem of large numbers of rows in a table. Instead of splitting existing rows across columns, an existing table is split into several structurally equivalent tables and the rows are portioned. While partitioning improves performance in many cases and makes large amounts

of data more manageable, it has also some drawbacks. Providing a consistent logical view on partitioned data sets often requires multiple join operations or even merge operations on application level. As a result, finding the partitions, both vertical and horizontal, is often not trivial and requires specific knowledge, how the data is accessed and used by the application.

The design of shards heavily influences the performance for finding and retrieving data. Thus, the partitioning strategy in use affects the system. Partitioning is usually realized using a partitioning key that allocates rows to shards. When *hash partitioning* is used, the result of a hash function applied to the key states which shard should be used. *List partitioning* provides a fixed mapping of keys to shards. Similarly, *range partitioning* assigns numerical ranges to shards. Combining different criteria is called *composite partitioning*. For instance, the aforementioned mechanism of consistent hashing can be considered as a combination of hash and list partitioning.

Although we explained partitioning using tables, rows and columns, most of the concepts are valid for non-relational database systems as well. A storage organization solely based on keys makes this concept even more apparent.

Replication, data partitioning and sharding represent orthogonal concepts, and they are partially contradictory. However, in large-scale database systems, all of these concepts are inevitable as fundamental mechanisms. Otherwise, systems could not be able to accept huge amounts of data and simultaneous read/write requests, deal with faults or provide low latency responses at the same time. Hence, deliberate trade-offs are required in practice.

6.3 Types of Distributed Database Systems

This section lists the major database system types that are in use for large-scale web applications. The general concept of each type is described and an exemplary product is introduced.

6.3.1 Relational Database Management System

The most common and predominant model for storing data is based on the idea of a relational model, introduced by Codd [Cod70] in the early seventies. The model stores data as tuples, forming an ordered set of attributes. In turn, a relation consists of sets of tuples. In terms of relational database systems, a tuple is a row, an attribute is a column and a relation forms a table. Tables are defined using a static, normalized data schema and different tables can be referenced using foreign keys. Structured Query Language (SQL) has established itself as generic data definition, manipulation and query language for relational data (e.g. SQL 99 [Eis99]). It has been adopted by almost all relational database management systems. Relational database implementations typically rely on the support of transaction and locking mechanisms in order to ensure atomicity, isolation, consistency and durability. These properties are essential for the relational model and can not be removed due to referential integrity and data consistency issues.

The great success of RDBMSs, especially for business applications, has led to increasing scalability requirements for many deployments over time. This not just includes scalability in terms of user load, concurrent queries and computational complexities of queries, but also in terms of plain data load. The traditional scale-up approach, using better hardware equipment, has absorbed further needs for growth for a certain time. But it soon became obvious that sooner or later, scaling out must be considered as well [Rys11].

The implications of the ACID paradigm combined with distributed systems make it very difficult to build distributed database systems based on the relational model. It requires the usage of complex mechanisms such as distributed transactions, that are feared by many developers, most of the time with good reason [Helo9, Alv11]. Enforcing ACID properties requires high complexity costs and in effect, they promptly hinder low latency and high availability. While replication and especially partitioning provide the basic tools for scaling out, it is the notion of distributed joins that makes distribution so painful. Join operations in a single instance database can be efficiently handled thanks to data locality. In case of distributed database systems, joins may require potentially large amounts of data to be copied between nodes in order to execute the necessary table scans for the join operations. This overhead renders such operations in many cases unusable. However, join operations are an inherent feature of the relational data model and can not be easily abandoned. On the other hand, the maturity of the relational model and several decades of experience make it still worth to spend extraordinary effort and apply complexity to scale out existing relational systems, as outlined by Rys [Rys11].

Apart from the big players from the commercial database world, MySQL[1] is a decent open-source RDBMS and it is very popular for web applications. It supports multiple backend storage engines, a broad subset of ANSI SQL 99 [Eis99] and several query extensions. Many web applications primarily struggle with read concurrency and scalability in terms of user load. Therefore, MySQL provides a simple yet powerful strategy using a master-salve architecture and replication. Incoming queries are then forwarded to instances according to the query type. Read-only operations (i.e. `SELECT`) are load-balanced to one of the slaves, while all other operations that contain write operations are forwarded to the master. Updates are then asynchronously replicated from the master to the slaves. This removes unnecessary load from the master, and helps to scale read operations. Obviously, it does not scale write operations. Master-slave setups can be further scaled using partitioning strategies and more complex replication setups [Sch08].

There also exists MySQL Cluster, a cluster variant of MySQL. It is based on a shared nothing architecture of nodes and uses synchronous replication combined with automatic horizontal data partitioning. Nodes can be either data nodes for storage, or management nodes for cluster configuration and monitoring. Due to the issues of distributed setups, MySQL Cluster has a very

1 `http://www.mysql.com/`

limited set of features compared to regular MySQL instances[1].

6.3.2 Non-Relational Database Management Systems

There is a plethora of distributed, non-relational storage systems. We outline four of the most popular types for large-scale web applications, although there are many others including RDF stores, tuple stores, object databases or grid-based storages.

Key/Value Stores

The idea of key/value-based storage system is very old and it relates to the concept of hash tables or maps in many programming languages. The storages allow to record tuples only containing a key and a value. While the key uniquely identifies an entry, the value is an arbitrary chunk of data and in most cases opaque for the database. In order to provide distribution and scalability, key/value stores apply concepts from distributed hash tables [Tano6]. The simple data model of key/value stores provides good scalability and performance. However, query opportunities are generally very limited, as the database only uses keys for indexes.

A very prominent storage system design based on the key/value principle is Dynamo from Amazon [DeCo7]. It incorporates several other techniques to provide a database systems that always allows writes, but may return outdated results on reads. This eventual consistency is tackled with vector clocks for versioning in case of partitions and application-level conflict resolution.

Redis[2] is an open-source key/value store that works in-memory, but supports optional persistence. As an advanced key/value store, it provides a data model for values and integrates publish/subscribe message channels. Persistence of key/value tuples can be achieved either using snapshots or by journaling. Redis supports master-slave replication that can be cascaded, resulting in a replication tree.

Document Stores

Document stores are similar to key/value stores. However, they require structured data as values using formats like XML or JSON. These values are referred to as documents, hence the name. Although the documents are using a structured format, there are often no fixed schema definitions. As a result, different documents with complex, varying structures can be stored in the same database, and structures of documents can evolve over time. Compared to key/value stores, document stores allow for more complex queries, as document properties can be used for indexing and querying.

1 http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-limitations.html
2 http://redis.io

A popular open-source document store is CouchDB[1]. It is written in Erlang and uses JSON as document format. CouchDB only provides an HTTP-based interface, which is inspired by REST. It makes use of different HTTP methods for create/read/update/delete operations on documents and query operations. CouchDB uses MapReduce-based functions for generating indexes over all documents. These functions, also written in JavaScript, allow the developer to produce indexes while iterating over the arbitrarily complex structures of each document. An exceptional feature of CouchDB is its powerful replication. CouchDB provides bidirectional replication, complex multi-master replication and is designed for offline usage with later data synchronization. As replication and concurrent writes may lead to conflicts, CouchDB applies an adapted variant of multiversion concurrency control [Ber81, Ree78]. As a consequence, conflicting writes lead to revision trees of document changes, that can be handled or merged later [And10]. Sharding is not a built-in feature of CouchDB, but there are external solutions that provide horizontal partitioning [And10].

Wide Column Stores

Wide column stores, sometimes also called sparse tables or column-oriented database systems, are database systems that store data by columns rather than by rows. Many wide column stores are inspired by BigTable [Cha06], a system designed by Google. BigTable is described as a "sparse, distributed, persistent multidimensional sorted map." The map character resembles key/value stores, but the keys are sorted. Persistence and distribution are obvious features of a distributed database systems. The traits of multidimensional and sparse are more important, as they define the essence of a wide column store. Multidimensional maps are essentially maps of maps, allowing nested data models. This concept is also known as column families. Sparseness describes the fact that a row can have arbitrary numbers columns in each column family, or even no column at all.

Besides the different data organization with deep structures and sparseness, the column-oriented storage has several impacts on database behavior. A major benefit is the efficiency of I/O operations during data access, when the column-oriented layout speeds up aggregation or column-based search/filter operations. As a column sequentially contains multiple values from different rows, efficient compression techniques can be applied.

In essence, wide column stores allow to efficiently store and query on very large, structured data sets. By reducing I/O overhead, queries can be executed faster and compression reduces storage requirements, compared to row-oriented systems. Thus, wide column stores are especially interesting for data warehousing and for big data sets, that must be queried. However, wide column stores have also several drawbacks. The column orientation increases the costs for insert and update operations, especially when not executed as bulk operations for multiple entries. A single insert/update results in multiple write operations in spread columns.

1 `http://couchdb.apache.org/`

Apache Cassandra[1] is an open source implementation of Google's BigTable [Cha06] that also incorporates several design principles of Amazon's Dynamo [DeC07] for fault-tolerance and data replication. Cassandra has been initially developed by Facebook, but has been released as open source. An interesting feature is its tunable consistency. By selecting quorum levels, consistency can be exactly chosen, ranging from aggressive eventual consistency models to strongly consistent blocking reads.

Graph Databases

Graph database systems are based on graph theory. They use graph structures for data modeling, thus nodes and edges represent and contain data. Nodes are often used for the main data entities, while edges between nodes are used to describe relationships. Both types can be annotated with properties. Graph databases have heavily benefited from the emergence of the Semantic Web [BL01] and the increasing popularity of location-based services. Both domains require data modeling with multiple relationships between entities, which becomes cumbersome in relational database systems. A notable strength of graph databases is the efficient traversal of data sets for certain queries. This permits, for example, fast queries for the shortest path between two nodes and other well-known graph-based computations. As opposed to many other database concepts, graph databases have no kind of primary index of all stored items, as the graph structure represents a purely associated data structure. That is why many graph databases employ external indexes in order to support text-based searches. Graph databases are also used for object-oriented persistence, as the notion of nodes and edges can be applied to objects and references. Graph databases thus circumvent the traditional object-relational impedance mismatch [Ire09] that occurs when object-oriented data is stored in relational database systems.

In terms of scalability, graph databases encounter a decent challenge. Sharding graph-based data sets means partitioning the graph structure onto different machines. As graphs are highly mutable structures, it is very difficult to find reasonable partitions that allocates graph nodes to available machines while minimizing traversal costs between remote nodes.

A popular open-source graph database for Java is neo4j[2]. It provides a native mapping of graph entities to Java objects and a transparent, disk-based persistence of these structures. Other features include transaction support, built-in graph traversers and graph algorithm implementations. Besides the native integration as part of a Java application, neo4j also provides a REST-based HTTP interface for data access. It ships with replication and (limited) sharding support.

1 http://cassandra.apache.org/
2 http://neo4j.org/

6.4 Summary

Large-scale web applications require solid backend systems for persistent storage. Essential requirements are availability, high performance and of course scalability for both read and write operations and for data volumes. These requirements cannot be addressed with a single database instance. Instead, distributed database systems are needed in order to support fault-tolerance, to scale out and to cope with highly concurrent read/write operations.

The CAP theorem challenges distributed database systems, as it rules out guaranteeed consistency, availability and partition tolerance at the same time. In distributed systems, failures cannot be avoided, but must rather be anticipated, so partition tolerance is virtually mandatory. Consequently, web applications are required to find a trade-off between strict consistency and high availability. The traditional ACID paradigm favors strong consistency, while the alternative BASE paradigm prefers basic availability and eventual consistency. Although exact consistency requirements depend on the actual application scenario, it is generally feasible to develop web applications with either one of them. However, the application must be aware of the consistency model in use, especially when relaxed guarantees are chosen and must be tolerated.

The internals of distributed database systems combine traditional database concepts with mechanisms from distributed systems, as the database consists of multiple communicating machines. This includes algorithms for consensus, distributed transactions or revisioning based on vector clocks. Replication is an important feature for availability and fault-tolerance. Partitioning addresses the need to handle large amounts of data and to allocate them to different physical nodes.

Relational database systems are built around the concept of relational data tuples and transactional operations. Although this model fits many business applications and can be used in various scenarios, there are other database concepts that provide different characteristics and their own benefits. These concepts include simple key/value data organization, relaxed or no schema definitions, column-oriented table alignment or the usage of graphs as the underlying data structure.

7 Recommendations

We have reviewed different concepts for concurrency and concurrent programming for components of a scalable web architecture in the previous three chapters. We will summarize our results by providing recommendations for each stage. As we have reviewed the components with varying details, also the scope of the advices differ. Understanding the implications of different server architectures helps to provide insights on the expected performance.

The application logic represents the core of a web application. It is usually the part of the architecture that contains the specific code implemented by own developers. Application code is often based on web frameworks or application containers. We limit our guidelines to concurrency concepts for this component. For distributed database systems, the impacts of different consistency guarantees and data models are in our main focus.

7.1 Selecting a Web Server Architecture

In chapter 4, we have considered mechanisms for I/O handling and connection concurrency for web servers. Popular architectures that incorporate different I/O and concurrency mechanisms have been reviewed. We have seen that sufficient performance result can be achieved with various models and there are solid implementations for most of them. In our architectural scenario, the web servers are challenged with highly concurrent connections, extreme I/O parallelism, but virtually not shared state. In this particular case, event-driven architectures using asynchronous I/O tend to provide better scalability under load, by requiring less memory and gaining better CPU utilization. Yet, this does not displace thread-based servers in general. When web servers are used in general-purpose setups, perhaps also executing the application logic in the same server instance, thread-based web servers are still a valid contender.

If there is the urgent necessity to implement a custom web server, the choice of the right concepts is often constrained by the platform and the programming language. If available, I/O multiplexing strategies (asynchronous/non-blocking) and cooperative scheduling should be considered. Also, low-level frameworks should be reviewed, as described in the evaluation part of chapter 4. They often implement the basic building blocks of a network server and allow the developer to focus on protocol-dependant and application-specific features.

7.2 Picking the Right Concurrency Concepts for Application Logic

The main focus of chapter 5 lies on concurrency concepts for application logic programming. In order to recommend appropriate concepts, we need to recap the different concurrency requirements that a web application might imply. On one hand, we might want to *reduce the latency* of request handling by parallelizing independent operations of a request. On the other hand, we might want to *coordinate different pending requests* in order to provide notification mechanisms and support server-initiated events. Furthermore, interactive and collaborative applications cannot rely on state that is solely isolated in the storage backend. Instead, some applications require to *share highly fluctuating and variable state* inside the application servers.

The reduction of latency of a request can mainly be achieved by parallelizing independent operations. For instance, parallel database queries and subdivided computations decrease the overall latency. An important property for accelerating requests is the ratio of CPU-bound and I/O-bound operations. Note that access to our platform components represents an I/O-bound operation. The latency of independent, CPU-bound operations can only be decreased by using more threads on more cores. When additional threads are used for heavy I/O parallelism, we roughly approach the same problem as seen previously for web servers. Using too many threads for I/O-bound operations results in decreasing performance and scalability issues due to context switching overhead and memory consumption. For thread-based programming models, the notion of futures or promises helps dispatching independent tasks and eventually collecting their results, without the need for complex synchronization. Actors can be used for I/O-bound and CPU-bound operations, although the efficiency depends on the underlying implementation. Event-driven architectures go nicely with primarily I/O-bound tasks, but they are entirely unusable for computationally heavy operations, as long as these operations are not outsourced to external components.

Coordinating requests and synchronizing shared application state are related. A first distinction is the scope of these operations. Some applications allow to partially isolate some application state and groups of users for interactivity. For instance, a browser multiplayer game session with dozens of players represents a conceptual instance with a single shared application state. A similar example is a web-based collaborative software application like a real-time word processor, running editing sessions with several users. When using session affinity, a running application instance can be transparently mapped to a designated application server. As a result, there is no need to share states between application servers, because each session is bound to a single server (a server can host multiple application sessions, though). In turn, the server can entirely isolate application state for this session and easily coordinate pending requests for notifications. In this case, event-driven architectures, actor-based systems and STM are appropriate concepts. The usage of locks should be avoided due to the risk of deadlocks or race conditions. Note that binding specific state to a certain server is contrary to our shared-nothing design of application servers.

In other cases, application state is global and cannot be divided into disjoint partitions. For instance, the instant messaging capabilities of a large social web application requires that any user might contact any other user. This leads to a scenario where state must be either outsourced to a distributed backend component (e.g. a distributed key/value store with pub/sub support such as redis), or it requires application servers to mutually share global state. The first variant works with all concurrency concepts. The latter is only applicable when a distributed STM or a distributed actor system is in use. Note however, these two approaches are contrary to our preferred shared-nothing style, as they introduce dependencies between application servers.

The conventional concept of a thread-based request execution model is still a valid approach when none of the aforementioned concurrency requirements are actually needed. In this case, the idea of a simple sequence of operations provides a very intuitive model for developers. If there is no imperative need to share state inside the application logic, dedicated backend storages should always be preferred. When retaining to a thread-based model and shared state inside the application logic is acutally needed, the usage of STM should be favored in order to prevent locking issues.

The actor model provides a versatile solution for multiple scenarios, but requires the developer to embrace the notions of asynchrony and distribution. Some concurrency frameworks such as akka[1] complement the actor model with other concepts such as STM and additional fault tolerance. This represents a strong foundation for distributed application architectures and should be considered when large scale is intended since the very beginning.

If the application logic of a web application primarily integrates the services provided by other platform components and does not require computationally expensive operations, single-threaded event-driven architectures are a good foundation. When used in a shared-nothing style, a sustaining scale-out can be accomplished by constantly adding new instances.

The actor model and single-threaded event-driven architecture share several principles. Both embrace asynchrony, queueing based on messages resp. events, and isolated state—either inside an actor or inside a single-threaded application instance. In fact, our web architecture combined with either one of these concepts for application logic resembles the original SEDA architecture [Wel01] to a great extent. Unlike SEDA, which describes the internals of a single server, we are then using very similar concepts for a distributed architecture.

7.3 Choosing a Storage Backend

We have considered several types of distributed database systems that can be used as storage backend for large-scale web applications in chapter 6. Assuming that a single database instance does not fit our requirements, due to missing opportunities to scale out and to provide high

1 http://www.akka.io/

availability, we need to choose a proper distributed database system for our applications. We have seen the trade-off between strong consistency and eventual consistency. The hype around non-relational database systems has led to a myriad of different database systems to choose from. In practice, the quest for the right system is often distorted by general arguments between the SQL camp and NoSQL camp by this time. The following guideline provides suggestions on how to choose the appropriate system.

The primary question is the *actual data model* imposed by the application. It is essential to focus on the intrinsic data model of the application first, before considering any database-specific models. It may also help to identify certain groups of data items that represent independent domains in the application. Additionally, it is helpful to keep in mind future requirements that might change the data model. For instance, an agile development style with frequent changes to the data model should be taken into account.

Next, the *scalability requirements* must be determined. Scaling out a blog application using additional database instances is a different undertaking than growing a large e-commerce site that already starts with multiple data centers. Also the dimensions of scale should be anticipated. Will the application be challenged by vastly parallel access to the persistence layer, or is it the fast-growing amount of data to store? The ratio of read and write operations can be important as well as the impact of search operations.

The third of these preliminary considerations aims at *consistency requirements* of the application. It is obvious that strong consistency is generally preferred by all parties involved. However, a review of the impact of stale data in different manifestations may help to define parts of the data model that can aligned with relaxed consistency requirements.

Due to their maturity, their features and our experiential basis, *relation database systems* still represent a strong foundation for the persistence layer of web applications. This is especially true, when the applications require strong consistency guarantees, transactional behaviors and high performance for transaction processing. Also, the expressiveness of SQL as a query language is conclusive and queries can be executed ad-hoc at any time. In order to tackle scalability for relational database systems, it is important to keep in mind functional and data partitioning strategies from the start. Sharding an unprepared relational database is extremely hard. But when denormalization and partitioning are designed into the data model, the opportunities for a sustainable scale-out are vastly increased.

Document stores are interesting as they provide a very flexible data modeling, that combines structured data but does not rely on schema definitions. It allows the rapid evolving of data models and fits very well to an agile development approach. The mapping of structured key-value pairs (i.e. documents) to domain objects is also very natural for many web applications, especially applications that are built around social objects. If database operations primarily involve create/read/update/delete operations, and more complex queries can be defined already in the development phase, document stores represent a good fit. The document-oriented data formats like JSON or XML are friendly for web developers. Moreover, many document stores allow

storing binary files such as images inside their document model, which can also be useful for web applications. Due to the underlying organization of documents as key/value tuples, document stores can easily scale horizontally by sharding the key space.

In general, *key/value stores* are the first choice when the data model can be expressed by key/value tuples with arbitrary values and no complex queries are required. Key/value stores not just allow easy scaling, they are also a good fit when many concurrent read and write operations are expected. Some key/value stores use a volatile in-memory storage, hence they provide unmatched performance. These systems represent interesting caching solutions, sometimes complemented with publish/subscribe capabilities. Other durable systems provide a tunable consistency based on quorum setups. Key/value stores that adopt eventual consistency often accept write operations at any time, even in the face of network partitions. As the data model resembles distributed hash tables, scaling out is generally a very easy task.

Graph databases are a rather specific storage type, but unrivaled when it comes to graph applications. Lately, social network applications and location-based services have rediscovered the strengths of graph databases for operations on social graphs or proximity searches. Graph databases often provide transaction support and ACID compliancy. When scaling out, the partitioning of graphs represents a non-trivial problem. However, the data model of such applications often tends to be less data-heavy. Existing systems also claim to handle several billion nodes, relationships and properties on a single instance using commodity hardware.

When designing very large architectures that involve multiple data centers, *wide column stores* become the systems of choice. They support data models with wide tables and extremely sparse columns. Wide column stores perform well on massive bulk write operations and on complex aggregating queries. In essence, they represent a good tool for data warehousing and analytical processing, but they are less adequate for transaction processing. Existing systems come in different flavors, either favoring strong or eventual consistency. Wide column stores are designed for easy scaling and provide sharding capabilities.

When designing large-scale web applications, also polyglot persistence should be considered. If no database type fits all needs, different domains of the data models may be separated, effectively using different database systems.

As an illustration, an e-commerce site has very different requirements. The stock of products is an indispensable asset and customers expect consistent information on product availabilities (⇒ relational database system). Customer data rarely change and previous orders do not change at all. Both types belong to a data warehouse storage, mainly for analytics (⇒ wide column store). Tracked user actions in order to calculate recommendations can be stored asynchronously into a data warehouse for decoupled analytical processing at a later time (⇒ wide column store). The implications for the content of a shopping cart is very different. Customers expect every action to succeed, no matter if a node in the data center fails. So eventual consistency is required in order to accept every single write (e.g. add to chart) operation (⇒ wide column store). Of course, the e-commerce application must then deal with the consequences of conflicting operations, by

merging different versions of a chart. For product ratings and comments, consistency can also be relaxed (⇒ document store). Very similar considerations are advisable for most other large-scale web applications.

7.4 Summary

The main criteria for web server architectures are the extent of I/O operations, the necessity to handle state, and the required robustness for connection concurrency. We have seen that it is advisable to consider I/O multiplexing strategies and cooperative scheduling for the web servers of our architecture.

For concurrency concepts as part of application servers, it is important to identify the actual concurrency requirements. Reducing the latency of requests requires parallelization strategies of the request logic. Coordinating different pending requests and sharing highly fluctuating and variable state inside application servers can either be handled using external backend components or high-level concurrency abstractions inside the application servers.

The choice for the appropriate storage backend depends on actual data model, the scalability requirements, and the consistency requirements of the application. Furthermore, polyglot persistence should be considered for large-scale web applications.

8 Discussion

We have seen how concurrency affects programming in different stages of a scalable web architecture. Also, the usage of distributed systems has been identified as an imperative for scalability, performance and availability. Let us now take a step back and reflect on the question, why concurrent and distributed programming is in fact so demanding and so different from conventional programming.

Conventional programming is based on the idea of a Von Neumann architecture. We have a single memory, a single processor and our program is a sequence of instructions. When the program is running, we have a single path of execution. State can be accessed and altered without thought, as it is exclusively used by our single flow of execution. Consistency can never be compromised. This is a valid abstraction for many programs and provides very clear implications and contracts. When the program or the machine crashes, the entire progress is basically lost, but we never have partial failures, except for programming errors.

Once we add additional flows of execution, multitasking is necessary for scheduling multiple activities on a single machine. As long as both flows do not share any state, the general abstraction remains unaffected. But if we allow state to be shared between activities and use preemptive scheduling, the abstraction becomes leaky. Although a path of execution is still processed sequentially, we cannot make assumptions on the order of interleavings. In consequence, different scheduling cycles yield different interleavings, which in turn affect the order of access of shared data. Consistency is at risk. It is especially the notion of a shared, mutable state that is ambiguously reflected in this model. When state is isolated by a single flow of execution, it can be modified without any risk. Once state is shared, its shared mutability becomes the main issue. Immutable state can be shared, as it does not represent a consistency hazard. If we still want to keep to our prior abstraction, we need to synchronize access to shared, mutable state using primitives such as locks. The moment we replace the processor with another one with multiple cores, the extent of nondeterminism vastly increases. We can now seize true parallelism, and multiple flows of execution can actually run in parallel physically. As a side note, this also changes the single memory assumption, because multiple cores introduce multiple levels of hierarchical caches, but most programmers will not be affected by this low-level modification. That's because this circumstance is entirely covered by operating systems, compilers and runtime environments.

Due to true parallelism, synchronization and coordination between flows of execution becomes imperative in case they share state or affect each other.

The conceptual idea of sequential execution can still be kept up, although it has only little in common with the actual execution environment of the hardware anymore. Especially when concurrent programming is used, the interleavings, overlappings and synchronizations of different flows of execution are not apparent. The inherent nondeterminism is not reflected in this model. When locking is used in a too coarse granularity, we end up in an execution model that is similar to the single core/processor system. Enforcing strict serializability eventually causes a sequential execution of all concurrent activities. When a sequential program runs on a single core machine, it is reasonable to have the notion of a program controlling its surrounding world entirely. When the application pauses, the environment does not change. True parallelism breaks this perception. The program has not anymore the sole control over the execution environment. Independently of any operations of a thread, other threads may change the environment.

As long as concise locking mechanisms or higher-level abstractions are used to protect from race conditions, shared state is still manageable and basically available for multiple activities running in parallel. Concurrency abstractions such as TM take away a lot of the actual complexity in place. By adding even more CPU cores, and more hardware resources, we can scale our application to provide additional throughput, parallelism, and gradually decrease latency.

However, at a certain point, this approach does not make sense anymore. On the one hand, hardware resources are limited—at least, physically. According to Amdahl's law, the maximum expected improvement by adding additional cores is also limited. On the other hand, at a certain scale, there are several non-functional requirements that become relevant: scalability, availability and reliability. We want to further scale our application, although we cannot scale a single node anymore. A crash still stops our entire application. But we might want the application to be resilient and to continue its service, even when the machine fails. The usage of multiple machines, spawning a distributed system, becomes inevitable.

The introduction of a distributed system changes the picture entirely. We now have multiple nodes that execute code at the same time, a further enhanced form of true parallelism. The most aggravating change is the notion of a state. In a distributed system, we don't have anything like a global state at first glance. Each node has its own state, and can communicate with remote nodes in order to ask for other states. However, whenever a state has been received, it reflects a past state of the other node, because that node might have already changed its state in the meantime. Also, communication with remote nodes has side effects that do not exist when accessing local state. This is not limited to unbounded latencies of message responses, it also includes the results of arbitrary message ordering. Furthermore, individual nodes may fail in a distributed system, or the network can become partitioned. This yields completely different failure models, as compared to local computing on a single machine.

It is obvious that our initial model—based on sequential computing—is completely broken now. The degree of complexity is many times higher than before. We have elemental concurrency

and parallelism, unbounded, unordered communication and neither global state nor time. As opposed to local calls, operations dispatched to remote nodes are asynchronous by nature. We are in need of new abstractions for tackling these inherent properties of a distributed system in order to build useful applications on top. There are generally two diametrically opposed approaches towards this challenge.

One approach aims for reconstructing as many previous assumptions and contracts as possible. Based on complex synchronization protocols, the notion of a global state is re-established and consistency becomes guaranteed again. By using coordination mechanisms, we can also fight against nondeterminsm down to enforcing the isolated, sequential execution of code fragmented to multiple nodes in an extreme case. RPCs restore the concept of function calls and allow to call remote functions implicitly and in the same way as local functions.

The other major approach accepts the essential flaws of distributed systems and incorporates them right into the programming model. This concerns primarily the acknowledgment of asynchrony and, as a result, the rejection of consistent global state. Eventually, asynchrony then exposes the inherent complexity of distributed systems to the programmer. This approach also favors explicit semantics rather than transparencies. Nodes have only local state and potentially stale states from other nodes, hence the notion of a global state is replaced by an individual and probabilistic view on state.

In a nutshell, the former approach hides complexity using very sophisticated and but often also laborious abstractions. However, by eluding asynchrony, this approach abandons some original gains of distribution in the first place. Enforcing synchrony requires a large coordination overhead, which in turn wastes lots of resources and capabilities of the distributed system. Also, when the provided abstractions are not appropriate, they make the development of distributed applications even more difficult. For example, when RPC abstractions pretend that remote entities are in fact local entities, the programmer cannot be aware of the consequences of inevitable network faults at runtime.

The latter approach exposes the complexities and forces the programmer to deal with them explicitly. Evidently, this approach is more challenging. On the other hand, by embracing asynchrony, failures and nondeterminism, high performance systems can be implemented that provide the required robustness, but also the true expressiveness of distributed applications.

As a matter of course, no approach is generally superior. Many of the concepts we have studied in the previous chapters tend to belong to either one of the camps. In fact, many existing distributed systems incorporate ideas from both approaches by applying high level abstractions when appropriate, but not renouncing complexity, when it is misleading.

9 Outlook

So far, we have considered different approaches towards concurrency as part of scalable architectures, including highly concurrent connection handling, concurrent application logic and distributed storage backends. We have become acquainted with several different concepts that address concurrency and scalability challenges and are increasingly established in the industry. Interestingly, many of these concepts are rather old, but have been rediscovered and revamped as existing solutions for arising problems in the era of cloud computing, big data and multi-cores.

In this chapter, we will go one step further and dare to take a glance at the future of web architechtures and concurrent programming in distributed systems. This overview is primarily based on ongoing efforts in the research community and emerging industry trends.

9.1 Emerging Web Architecture Trends

Web architectures have been considered in chapter 3. The entire architecture is designed around the essence of the HTTP protocol. If the protocol changes over time, this may also affect the architecture. Furthermore, the design and the internals of the components of a web architecture may be influenced by other trends as well.

9.1.1 The Future of the HTTP Protocol

When the web was getting increasingly popular in the the mid and late nineties, the final HTTP/1.1 specification was published under a lot of pressure. This has led to several inaccuracies and ambiguous statements and artifacts in the resulting standard [Not12]. Consequently, the IETF constituted the HTTPbis Working Group[1], which is responsible for maintaining the standard. The working group not just collects problems with the current standard, but also revises and clarifies the specification, especially with regard to conditional requests, range requests, caching and authentication. It is planned that these improvements eventually transition into the HTTP/2.0 specification.

1 http://tools.ietf.org/wg/httpbis/

Extension Protocols to HTTP/1.1

Besides misconceptions in the HTTP/1.1 specification, other problems and weaknesses of HTTP/1.1 have been manifested in the field. Common points of criticism include performance issues, high latencies, verbose message formats and weak support for extensions like custom HTTP authentication mechanisms. The rigid client-initiated request/response cycle has also been criticized, because highly interactive web applications often require server-initiated communication as well. This has not just been addressed in the WebSocket protocol [Fet11] recently, but also by specific HTTP protocol extensions such as Full-Duplex HTTP [Zhu11]. Similar HTTP protocol extensions like HTTP-MPLEX [Mat09] integrate multiplexing and header compression into the protocol.

The waka Protocol

Fielding, who has introduced the REST architectural style [Fie00], has also identified drawbacks of HTTP when using REST as an integration concept for large enterprise architectures. Drawbacks include head of line blocking of pipelined requests and legacy issues with verbose message headers. Also, Fielding denotes the absence of unsolicited responses and better messaging efficiency, especially for low-power and bandwidth-sensitive devices. Consequently, Fielding has started to work on waka[1], a token-based, binary protocol replacement for HTTP. It is deployable via HTTP, using the `Upgrade` header and introduces new methods on resources like `RENDER` or `MONITOR`. Request and transaction identifiers are used to decouple request and response messages, allowing more loosely coupled communication patterns. Waka can be used with different transport protocols and it is not limited to TCP. Besides binary communication, waka uses interleaved messages for better performance. Fielding is still working on the waka specification, and there is no draft available yet.

SPDY

Other efforts for more efficient web protocols have been expended in the industry. The most popular initiative is led by Google and works on an application-layer protocol called SPDY [Bel09]. SPDY focuses on efficient transporting of web content, mainly by reducing request latencies. In HTTP/1.1, performance (i.e. throughput) can be increased by using multiple persistent connections and pipelining. According to the SPDY protocol designers, this connection concurrency is responsible for increased latencies of complex web sites. They argue that a single connection between the client and the server is more efficient, when combined with multiplexed streams, request prioritization and HTTP header compression. Advanced features of SPDY include server-initiated pushes and built-in encryption. SPDY is designed for TCP as the underlying transport

1 `http://tools.ietf.org/agenda/83/slides/slides-83-httpbis-5.pdf`

protocol. SPDY is already in use by Amazon, Twitter and Google services, and the browsers Google Chrome and Mozilla Firefox provide client-side protocol implementations. Microsoft has suggested an alternative HTTP replacement, called HTTP Speed+Mobility [For12]. It incorporates concepts of SPDY and the WebSocket protocol, but it is enriched with optional protocol extensions for mobile devices.

HTTP/2.0

At the time of writing in early 2012, the HTTPbis Working Group has been rechartered[1] and is about to start work on an upcoming HTTP/2.0 draft [Not12]. We have seen recurring concepts in the different protocols. Fortunately, most of the ideas have been acknowledged by the HTTPbis group and their protocol designers are in close dialog. Thus, the aforementioned protocols may influence the design of HTTP/2.0 and boost the HTTP/2.0 specification.

9.1.2 New Approaches to Persistence

The NoSQL hype has already been shaking the database world and has led to a more versatile toolbox for persistence in the mind of developers and architects. Still, relational database systems are the most prominent and popular choices for persistence. However, there is some general criticism on the essence of current RDBMSs.

Future Architectures of Relational Database Management Systems

Stonebraker et al. [Sto07] point out that RDBMSs are still carrying the legacy architecture of the first relational database systems such as System R. These systems have been designed mainly for the business data processing at that time, and not as a general-purpose solution for all kinds of persistence. Also, different hardware architectures prevailed at that time, and interactive command line interfaces for queries constituted the primary user interface for database access. In order to provide high performance and throughput on machines available back then, traditional concepts such as disk-based storage and indexing structures, locking-based concurrency control and log-based failure recovery have been developed and implemented. Latency has been hidden by extensive use of multithreading. Although these concepts have been complemented with other technologies over time, they still represent the core architecture for each RDBMS available. Stonebraker argues that this architecture is not appropriate anymore. It is especially not appropriate, when RDBMSs are used in a "one-size-fits-all" manner for many different kinds of persistence applications. According to Stonebraker, a "complete rewrite" is necessary in order to provide high performance architectures for specialized database applications, with distinct requirements. Only a rewrite would allow to get rid of architectural legacy concepts and to realign on hardware trends

1 http://lists.w3.org/Archives/Public/ietf-http-wg/2012JanMar/0098.html

such as multi-core architectures and large main memories. Furthermore, Stonebraker campaigns for a better integration of database access and manipulation as part of the programming model without exposed intermediate query languages like SQL. In their work [Sto07], they introduce a prototype for such a new database system. The system runs multiple single-threaded instances without any communication and resides in main memory. It is still a row-oriented relational database and provides full ACID semantics. It appears that not just NoSQL databases will provide more specialized solutions for different usage scenarios. If Stonebraker ends up being right, we will also see further diversification of relational database management systems and engines for online transaction processing.

In-Memory Database Systems

The idea of in-memory database systems [Gm92] and hybrid in-memory/on-disk systems becomes increasingly popular, as the performance increases of new CPUs, main memory components and harddisk technologies tend to drift apart. When low latencies are required, for example for interactive and real-time applications, in-memory database systems represent an attractive solution. When durability is required, they can be combined with asynchronous, non-volatile disk-based persistence.

Event Sourcing and Command Query Responsibility Segregation

And others again generally raise to question the way we are handling and persisting mutable state in applications. An alternative paradigm, which is increasingly receiving attention, is the combination of two patterns: event sourcing [Fow05] and Command Query Responsibility Segregation (CQRS) [Dah09].

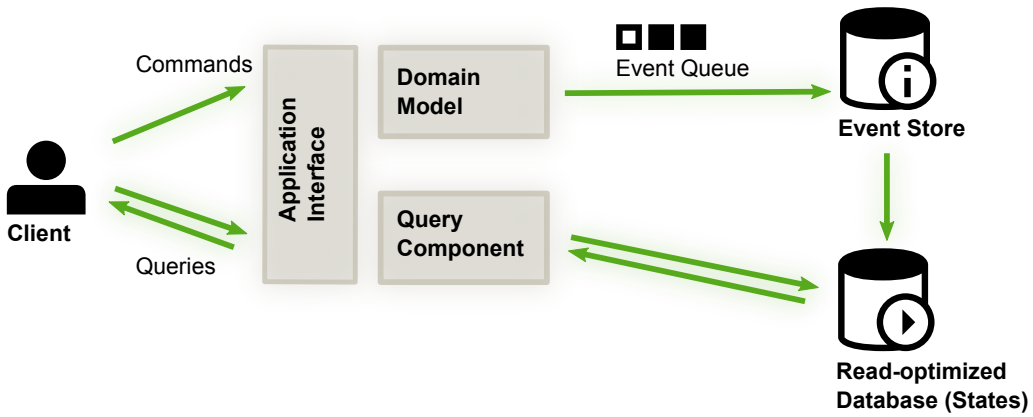

Figure 9.1: A very simplified illustration of the flow control in an architecture that uses CQRS and event sourcing.

The underlying idea of event sourcing is to capture all state changes explicitly as domain events. Thus, the current application state becomes the result of the sequence of all events captured so far. Events are stored using a durable event log (e.g. a database). Instead of changing and updating existing state in a database, new events are emitted when application actions take place. This has several benefits. For instance, listeners can easily be added to the stream of events and react to specific events or group of events in an asynchronous manner. The main benefit of event sourcing is the ability to replay state evolution. All events are persisted, hence the application can rebuild any prior state by reapplying the events from the event log. Even alternative orderings or the effect of injected additional events can be analyzed. Event sourcing also supports snapshotting, since events natively represent incremental updates.

Traditional persistence layers provide a single model for reading, reporting, searching and transactional behavior. The CQRS pattern decouples different concepts, namely command operations and query operations, using separate models. This separation of concerns improves scalability and integrates an eventually consistent behavior into the application model.

When both patterns are combined, as shown in figure 9.1, command operations emit new events, which are added to the event log. Based on the event log, the current application state is built and can be queried, entirely decoupled from commands. This encapsulated approach provides interesting scalability properties and may find its way into future web architectures.

9.1.3 Tackling Latencies of Systems and Architectures

Coping with latencies is one of the big issues of large-scale architectures and distributed systems. Increasingly complex multi-core CPU architectures with multiple cache levels and sophisticated optimization mechanisms have also widened the latency gap of local operations. Table 9.1 shows the vast differences of latencies of various local and remote operations. When designing and implementing low latency systems, it is inevitable to take into account these numbers—both locally and for distributed operations. The fact that hundreds of machines of a web architecture may work together for responding to a single request should not be obvious for the user just by yielding high latencies.

The classical model of a Von Neumann architecture running sequential executions may still provide a theoretical abstraction for programmers, but modern hardware architectures have slowly diverged from it. For tackling latency locally, it is important to understand the implications and properties of modern CPU architectures, operating systems and runtime environments such as the Java Virtual Machine (JVM). This notion, sometimes referred to as "mechanical sympathy" [Tho11], has renewed interest in approaches like cache-oblivious algorithms [Fri99]. These algorithms take into account the properties of the memory hierarchy of a CPU. They favor cache-friendly algorithm designs over algorithms that solely reflect computational complexity. A recent example is the Disruptor [Tho11], which is a high performance queue replacement for data exchange between concurrent threads. The Disruptor basically uses a concurrent, pre-allocated ring buffer with custom barriers for producer/consumer coordination. It does not use locks and

Operation	Latency
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes w/ cheap algorithm	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Table 9.1: Common operations and their average latencies. Note that these numbers are exemplary and platform dependent. Yet, they give a rough round-up on the impact of different operations. Source: Talk "Building Software Systems at Google and Lessons Learned" by Jeffrey Dean (Google) at Standford (`http://goo.gl/0MznW`).

heavily promotes CPU caching.

The latency of remote operations is more difficult to tackle. Network latencies are generally bounded by physical constraints. Of course, efficient infrastructure layouts for data centers are a prerequisite. However, Rumble et al. claim that "it's time for low latency" for the network [Rum11] in general. They demonstrate that between the years 1983 and 2011, network latencies have improved far more slowly ($\sim 32x$) than CPU speed ($> 1{,}000x$), memory size ($> 4{,}000x$), disk capacity ($> 60{,}000x$) or network bandwidth ($> 3{,}000x$). According to their analysis, this is mainly caused by legacy technology stacks. Rumble et al. argue that round trip times of $5 - 10\mu s$ are actually possible in a few years. New network interface controllers already pave the way for such latencies, but current operating systems still represent the major obstacle. Without an entire redesign, the traditional network stacks are not capable of such low latencies. Rumble et al. also note that this change requires new network protocols that are aware of such low latencies. Once available, low latency networking may partially influence distributed computing, as it bridges the gap between local and remote operations and provides better temporal transparency.

9.2 Trends in Programming Languages

Programming languages have always been heavily influenced by programming paradigms, which in turn have been characterized by general computing trends. The cumbersomeness of low-level machine code yielded imperative programming languages. These languages take advantage of compilers or interpreters in order to generate low-level machine code based on easier to handle

higher-level languages. As a result of increasing complexity and scale of programs, there was a need for finer granularity and encapsulation of code, which led to new modularization concepts. This has been later complemented and extended by the object-oriented paradigm, which introduced new concepts like polymorphism and inheritance. The object-oriented paradigm promotes the design and implementation of large, but manageable, software systems and thus addresses the requirements of large-scale applications. However, the prevalence of multi-core architectures and the pervasion of distributed systems and applications in everyday life represent other trends affecting upcoming programming languages. Some even believe that "the concurrency revolution is likely to be more disruptive than the OO revolution" [Sut05]. Although this is a controversial statement, it is remarkable that most of the new programming languages take concurrency seriously into account and provide advanced concurrency concepts aside from basic threading support [Gho11].

Apart from the object-oriented paradigm, there are several less common paradigms such as declarative or functional programming that focus on high expressiveness. Programming languages following these paradigms have been considered as esoteric and academic languages by the industry for a long time. Interestingly, there is an increasing popularity of these alternative concepts, especially in functional programming and even for web application programming [Vin09]; not least because these languages provide inherently different concurrency implications. As functional languages favor immutability and side-effect free programming, they are by design easier to execute concurrently. They also adapt other techniques for handling mutable state and explicit concurrency.

The gap between imperative, object-oriented languages and purely functional languages has been bridged by another tendency: multi-paradigm programming languages. By incorporating multiple paradigms, programming languages allow the developer to pick the right concepts and techniques for their problems, without committing themselves to a single paradigm. Multi-paradigm languages often provide support for objects, inheritance and imperative code, but incorporate higher-order functions, closures and restricted mutability at the same time. Although these languages are not pure in terms of original paradigms, they propose a pragmatic toolkit for different problems with high expressiveness and manageable complexity at the same time.

The higher expressiveness is often a basic prerequisite for the design of so-called domain-specific languages [Gho10], another trending topic in software development. Domain-specific languages allow to specify and express domain objects and idioms as part of a higher-level language for programming. By providing a higher level of abstraction, domain-specific languages allow to focus on the application or business domain while concealing details of the programming language or platform. Several frameworks for web development can be considered as domain-specific languages for the domain of web applications.

9.2.1 New Programming Languages for Web Programming

Thanks to the web, JavaScript has not just become the *lingua franca* of the web, but also the most widespread programming language in general. Every browser, even mobile ones, act as an execution environment for JavaScript applications, thus JavaScript is available on virtually all computing devices. But JavaScript is also increasingly popular outside the web browser, thanks to projects like node.js. Microsoft uses JavaScript as the main programming language for Metro applications in the upcoming Windows 8 release. JavaScript is a multi-paradigm language that incorporates prototypal object inheritance combined with many functional aspects of Scheme. It has been designed as a general-purpose programming language, but reached attention, and sometimes also faced hatred[1], not until it became popular through the web. However, JavaScript is notoriously known for some of its "bad parts" [Cro08]. Deficiencies include odd scoping, global variables, automatic syntax correction (i.e. semicolon insertion) with misleading results, and problems with the type system and with value comparisons.

As we are somehow locked-in to JavaScript concerning browsers, there are several approaches to circumvent these drawbacks, as long as they are not fixed in the language itself by upcoming specifications. Crockford suggests a subset of the languages that makes only use of the "good parts" of the language [Cro08]. Others attempt to transcompile (i.e. executing source-to-source compilation) different languages to JavaScript. Popular examples therefore are ClojureScript [McG11] and CoffeeScript[2]. ClojureScript translates Clojure code into JavaScript, though some of the Clojure features are missing. For instance, JavaScript is single-threaded, so the usage of concurrency concepts of Clojure is limited. CoffeeScript takes a different route. It is a language that exclusively transcompiles to JavaScript and has been designed as a syntactic replacement for JavaScript. CoffeeScript not just adds syntactic sugar, but also provides some advanced features like array comprehension and pattern matching.

When Google was dissatisfied with the progress on new JavaScript specifications and was reasoning about the future of web programming, they identified the need for a general-purpose web programming language for both clients and servers, independent of JavaScript. This need was shortly after addressed by Google's new Dart [Tea12] programming language. Dart is derived from JavaScript, but incorporates several concepts from Java and other languages. It is class-based, like Java, and supports interfaces, abstract classes and generics. Dart is dynamically typed, but annotations can be used to enforce static typing. A core library provides common data structures and operations, and a DOM library supports HTML5 DOM. For server applications, Dart includes an I/O library with an asynchronous, non-blocking programming model and an event loop. Dart also ships with an HTTP library as a foundation for web servers. Concerning concurrency in Dart, the specification disallows shared-state concurrency. However, Dart proposes actor-like structures, so-called isolates. Each isolate represents an independent flow of control and it can

1 http://www.crockford.com/javascript/javascript.html
2 http://coffeescript.org/

thus be assumed to be single-threaded for the developers. Multiple isolates can communicate via message passing. There are different ways to execute Dart applications. Google's Chrome browser already supports Dart. Time will tell if other browser vendors will eventually support Dart as well. As an interim solution, there is a Dart-to-JavaScript transcompiler that generates pure JavaScript code out of Dart sources. For usage outside of the browser, Google provides a separate Dart virtual machine.

Dart and node.js demonstrate the possibility of natively using the same programming language for client-side and server-side web application development. Some node.js web framworks already support the usage of the same functions both for browsers and the server. For example, the same template rendering functions can be used, both for complete page generation on the server and for partial updates in the browser. Also RPC-like frameworks have emerged that allow the client-side application to execute remote operations on the server, entirely using JavaScript.

Other frameworks like the Google Web Toolkit[1] or Vaadin[2] address the desire for a single programming language by implementing web applications in Java, then generating client-side code automatically.

Opa[3] is a dedicated approach which takes this idea one step further. Opa combines a new OCaml-inspired programming language, a web application framework and a runtime platform consisting of a web server, a database and a distributed execution engine. All parts of a web application can thus be implemented entirely in Opa, using a single language and a single programming model. For deployment, the code is automatically sliced into different compiled components: JavaScript code for the browser, native code for the platform and generated database queries/scripts. The Opa language focuses on a strong and static type system in order to provide security and to prevent traditional web security flaws such as SQL injections or cross-site scripting. The platform claims to provide a scalable architecture based on an event-driven, non-blocking concept and messaging mechanisms similar to Erlang. Opa also supports distribution by using multiple machines.

A downside of concepts such as Opa is, for instance, the increasing overhead of development cycles due to steady compilation. Also, the thorough unification of different conceptual layers hinders flexibility and makes it more difficult to integrate custom components or legacy applications.

9.2.2 The Rise of Polyglot JVM-based Languages and Virtual Machines

In the Java programming language, Java source files are compiled to intermediary class files, using a byte code format. Class files can then be executed by a JVM. This indirection was primarily

1 https://developers.google.com/web-toolkit
2 https://vaadin.com/home
3 http://opalang.org/

chosen for platform independence, as the same compiled class file can be executed on multiple architectures using the proper JVM ("write once, run anywhere"). Platform and architecture specificities are thus only part of the JVM implementations.

The separation of source code language and byte code format executed by JVMs has led to another trend: alternative languages that compile to the JVM byte code. With the introduction of `invokedynamic` [Ros11] in Java 7, the support for dynamically typed languages has been greatly increased [Tha10]. Most of the popular scripting languages and many other programming languages are now available for the JVM, as shown in table 9.2. For web application development, this allows the usage of frameworks very similar to Ruby on Rails, but hosted on the JVM. Erjang[1] converts binary Erlang beam files to Java class files and executes them in the JVM.

There is increasing criticism addressing drawbacks of the Java programming language. New features and enhancements are said to become integrated too slowly. For instance, the support for closures (Project Lambda [Goe10]) has been postponed several times. Dissatisfaction with the state of Java on the one hand, the prevalence of the powerful JVM on the other hand have given rise to entirely new programming languages exclusively targeting the JVM. These languages often combine common Java concepts with advanced programming features, or they even support completely different programming paradigms. Hence, most of the languages are in fact multi-paradigm programming languages. Furthermore, they often provide higher-level concurrency abstractions apart from traditional Java multithreading. Table 9.3 lists some of the most popular JVM languages aside from Java.

Groovy[2] incorporates concepts from Python, Ruby, Perl, and Smalltalk. It is a dynamic scripting language for the JVM. With GPars, Groovy provides a very advanced concurrency library. It supports many of the concepts we have seen, including actors, dataflow concurrency constructs, STM, agents, Fork/Join abstractions, asynchronous background tasks, and concurrent/parallel data structures.

Scala is a multi-paradigm language built on functional and object-oriented concepts. We have already seen in chapter 5 that Scala supports the actor model for concurrency. As Scala

JVM Language	Inspired By
Rhino	JavaScript
Jython	Python
JRuby	Ruby
Jacl	Tcl
Erjang	Erlang

Table 9.2: Examples for programming languages that have been ported to the JVM.

1 `http://erjang.org/`
2 `http://groovy.codehaus.org/`

Language	Main Paradigms	Type System	Main Concurrency Model
Java	object-oriented, imperative	static, strong	Threads
Groovy	object-oriented, functional	dynamic, strong	various concepts (GPars)
Clojure	functional	dynamic, strong	STM, Agents
Scala	functional, object-oriented	static, strong	Actors
Fantom	object-oriented, functional	static/dynamic, strong	Actors
Kotlin	object-oriented, functional	static, strong	Threads (Actors soon)
Ceylon	object-oriented, imperative	static, strong	Threads

Table 9.3: Popular stand-alone languages for the JVM.

also supports the usage of Java APIs, also low-level threading is possible. So far, Scala is the most popular general replacement language for Java and gains a foothold also in enterprise environments. Clojure is a Lisp dialect for the JVM with a strong focus on concurrency and functional concepts, as seen in in chapter 5. Ceylon[1] is a Java-inspired language that is designed for large application programming. Kotlin[2] is a general-purpose language for the JVM with a strong emphasis on concise code and typing. Kotlin uses regular Java threads for concurrency. Fantom[3] incorporates object-oriented principles enhanced with mixins (partially implemented interfaces), functional concepts and varying typing principles. Fantom provides strong concurrency support by using the actor model and message passing. Additionally, Fantom supports the notion of immutability as core concept of the language.

Most of the alternative JVM languages allow the usage of the standard Java library components, either directly or indirectly via proxy constructs. Consequently, many libraries (e.g. database drivers) originally designed for Java can be utilized. This also works vice versa in several cases, when components developed in a non-Java language can be integrated into Java applications, thanks to exported Java interfaces. The byte code compatibility of the languages does not just allow to run application components developed with different languages inside the same JVM. It also enables the gradual redesign of legacy applications into new languages, without changing the underlying platform.

But the concept of virtual machines for byte code execution is not limited to Java. Microsoft's .NET platform sets similar objectives with their Common Language Runtime (CLR). The separation of a virtual machine and different programming languages that compile to the same byte code for that virtual machine provides independence of hardware architectures, general availability and versatility. The virtual machine designers can thus focus on efficient execution and performance optimizations on various architectures. At the same time, programming languages can be designed that strive for higher abstractions and incorporate advanced language concepts.

1 http://ceylon-lang.org/
2 http://www.jetbrains.com/kotlin/
3 http://fantom.org/

This is particularly interesting for web application development, where "bare metal" coding is not required.

The recently initiated EU project RELEASE[1] evaluates the future usage of Erlang's actor model for architectures with massively parallel cores. This primarily addresses the Erlang virtual machine, which currently only scales to a limited number of cores available. Especially the mapping of huge numbers of actors mapped to hundreds of cores and the performance of message passing between actors has to be estimated. The project also covers distribution and deployment concerns and the capability of the virtual machine to build heterogeneous clusters.

RoarVM[2], another academic project related to virtual machines and concurrency, is a manycore virtual machine for Smalltalk. Ongoing research evaluates how various concurrency models can be implemented efficiently on a single virtual machine [Mar12] for manycore systems.

9.2.3 New Takes on Concurrency and Distributed Programming

Chapter 5 outlined most of the prevailing approaches towards concurrency. Many emerging programming languages pick up one of the concepts that provide higher abstractions than low-level multithreading. Concurrent and distributed programming is about to become one of the biggest challenges of our time to be faced for computing in general. As a result, there is still much ongoing research effort in finding programming models that tackle concurrency and distribution more naturally. While some specifically target multi-core concurrency, others address concurrency more generally as an intrinsic property of distributed computing. We now take a look at some of these concepts and study their original ideas.

Harnessing Emergence and Embracing Nondeterminism

Ungar et al. [Ung10] suggest a programing model for multi-core systems that embraces nondeterminism and supports emergence. It follows the example of natural phenomenons such as bird flocks or ant colonies. In these systems, a large number of entities interact based on a common set of rules, each one with an individual view of the world and in an asynchronous style. Still, the systems show coordinated and robust behavior without any kind of explicit synchronization among entities, which is known as emergence. Ungar et al. argue that existing approaches towards concurrency focus too much on taming indeterminacy instead of accepting it.

Ungar et al. integrate the notions of emergence into an extended object-oriented programming concept, by introducing so-called ensembles and adverbs as first-class language concepts. Ensembles are essentially parallel collections of objects, that can be referenced as a single entity by the outside world. Messages sent to an ensemble are dispatched to all of its members in parallel.

1 http://www.release-project.eu/
2 https://github.com/smarr/RoarVM

Invocations in object-oriented programming are defined by the tuple of caller, callee and invocation arguments. For ensembles, this tuple is extended by adverbs that determine additional call semantics. An adverb describes, which or how many members of an ensemble are involved, how invocations are propagated and how results are collected and returned. Based on these concepts, a large numbers of objects can interact without explicit application-level synchronization. Ungar et al. have implemented an experimental virtual machine and a JavaScript-inspired language for evaluation. They realized that many traditional algorithms are not compatible with this model and have to be redesigned entirely. Also, they identified several ambiguities in the language semantics of ensembles that they want to address in their future work.

Memories, Guesses and Apologies for Distributed Systems

In their paper "Building on Quicksand" [Hel09], Helland and Campbell campaign for a paradigm shift for building large-scale, fault-tolerant systems. Due to increased scale and complexity of systems, the scope of failures to deal with has reached critical dimensions. While transparent reliability in confined hardware components (e.g. mirrored disks) appears appropriate, the notion of transactional behavior and synchronous checkpointing for fault-tolerance in large, distributed systems is too complex to be managed anymore, according to Helland and Campbell. They also observe that asynchronous checkpointing in order to save latency can not be implemented without losing a single, authoritative truth as part of the distributed system. With due regard to the CAP theorem, they favor a different approach that can also be applied to distributed programming in general.

Instead of relying on traditional ACID properties and serializability, they prefer eventual consistency as an integral part of the application itself. Therefore, they introduce the concept of memories, guesses and apologies as well as probabilistic rules for application programming. Memories are the local views that each node of the system has at a given time. Obviously, these views can differ between nodes. Guesses describe the notion that each action of a node is not based on a global view, but only on its own memories. Ideally, these memories resemble the "global truth" of the system as much as possible. However, actions may be executed based on wrong guesses. Resulting mistakes must be handled by apologies, either automatically by the application, or involving humans. Making guesses and handling apologies is based on probabilistic properties and captured by predefined rules.

Furthermore, Helland and Campbell outline how carefully designed application operations can facilitate this approach, when they provide the following semantics: associative, commutative, idempotent and distributed. These properties inherently allow the reorderability and repeatability of operations in the system. They make systems also more robust and resilient to typical failures in message-oriented distributed systems [Hel12]. Helland and Campbell favor these operations instead of explicit mutable states for application design. They demonstrate how traditional business applications, including bank accounting, can be described and realized using this approach.

Consistency and Logical Monotonicity

The group of Hellerstein addresses the challenge of consistency and parallelism in distributed systems by applying declarative/logic programming and monotonicity analyses [Hel10]. They believe that declarative database query languages, that are able to parallelize natuarlly, can provide an appropriate programming paradigm for distributed and concurrent systems, when combined with temporal awareness.

The idea is to accept eventual consistency whenever possible, but identify locations where the lack of strong consistency results in unacceptable consistency bugs. Therefore, they introduce the notion of consistency as logical monotonicity. Based on the theory of relational databases and logic programming, a program can be defined as monotonic or non-monotonic. A monotonic program incrementally produces output elements, never revoking them at a later time due to further procressing. For instance, a projection on a set is a monotonic operation. Instead, non-monotonic operations require the entire processing to be completed in order to produce outputs. Hence, non-monotonic operations are blocking operations. Aggregation or negation operations on sets are examples for non-monotonic operations.

Applied to distributed computing, we can also differentiate monotonic and non-monotonic distributed operations. Monotonic operations do not rely on message ordering and can tolerate partial delays of messages. Instead, non-monotonic operations require coordination mechanisms such as sequence numbers or consensus protocols. For instance, everything that involves distributed counting is a non-monotonic operation with the need for coordination and waiting. Hellerstein et al. further show that monotonic operations guarantee eventual consistency, independent of the order of messages, while non-monotonic operations require coordination principles in order to assure consistency. Based on a declarative language, consistency behaviors can be analyzed and non-monotonic locations can be identified automatically. These locations are so-called points of order, that need to be made consistent by adding coordination logic.

Hellerstein et al. implemented a prototype [Alv11] based on an underlying formal temporal logic concept. The purely declarative prototype uses a domain-specific subset of Ruby due to syntax familiarity.

Distributed Dataflow Programming

Declarative dataflow programming provides a concurrency model with inherent coordination, entirely hidden from the developer. Massively parallel data-centric computing frameworks such as MapReduce [Dea08] have shown the strong points of dataflow programming. However, programming abstractions like MapReduce heavily constrain the expressiveness compared to pure, non-distributed dataflow languages. Thus, only a small amount of existing algorithms can be applied for MapReduce-based computations. Combining an expressive programming model including dataflow concurrency with a scalable and fault-tolerant distributed execution engine represents a sweet spot for programming in the large.

One approach that addresses this demand is the Skywriting scripting language [Mur10] and the CEIL execution engine [Mur11]. Skywriting is a language that resembles JavaScript syntactically, but is purely functional by design. As opposed to other data-centric computation languages such as MapReduce, Skywriting is Turing-powerful. It provides support for (sub-)task spawning, explicit future references and a dereference operator. Task spawning yields future references that can be dereferenced later on, providing implicit synchronization. Task execution is idempotent and deterministic, so that functional purity is not comprimised.

The underlying CEIL execution engine provides cooperative task farming and implicitly generates acyclic graphs of task dependencies. It also schedules tasks to available worker machines and provides transparent coordination and synchronization. A set of rules enforces schedulability by managing the dependencies between spawned tasks and their inputs and outputs. For instance, the dependency graph can not contain cycles, nor can a task become orphaned. The execution engine also provides fault-tolerant features including re-execution of tasks in case of worker crashes and master replication for master fail-overs.

Murray et al. argue that the Skywriting language combined with the CEIL engine allows the implementation of imperative, iterative and recursive algorithms that run on large clusters. Thanks to declarative concurrency, the implementers of algorithms do not have to reason about coordination and synchronization.

Functional Relational Programming

Brooks and Frederick [Bro87] identify complexity as one of the four fundamental difficulties of software engineering next to conformity, changeability and invisibility. Complexity is further divided into essential and accidental complexity. Accidental complexity is complexity that we create by ourselves as part of a solution when building systems. In contrast, essential complexity is an inherent part of the problem to be solved and independent of the solution.

Moseley and Marks [Mos06] take up again the notion that complexity is the single major challenge for large-scale applications. However, they disagree with Brooks and Frederick by rejecting the statement that most of the complexity of systems is essential. Instead, Moseley and Marks argue that it is state, that is primarily responsible for most of the (accidental) complexity. There are secondary properties like flow of control and expressiveness, but they directly relate to state. Thus, Moseley and Marks reflect on new ways how state can be limited and managed in order to simplify large-scale systems. They propose essential logic, essential state and accidental state and control as the three components of a simpler architecture. Essential logic ("behavior") is business logic, that is not concerned with state at all. It only defines relations, pure functions and integrity constraints. Essential state ("state") is a relational definition of stateful components of the system. It is defined by schemata for system inputs and subsequential state updates and state changes. Accidental state and control is a specification, where state should be used and what controls should be applied. However, it does not affect the logic. So interactions with the system result in changes of essential state, which in turn may trigger actions in the other

components. As a result, the essential logic may execute operations that affect the system output. For implementations, Moseley and Marks suggest a relational, logic-driven programming model for most of the parts. Pure, functional programming can be added for some parts of the essential logic.

This idea, also known as functional relational programming, represents an extreme approach on how to isolate and handle state and it does not resemble any of the popular programming models. It rejects the notion of coupling state and behavior, that characterizes object-oriented programming. Instead, functional relational programming takes an entirely declarative route, mainly resting upon relational principles and to minor extend the usage of side-effect free functional programming constructs. It is still uncertain, whether this model will soon immerse into mainstream development models for application programming. However, the strong focus on declarative state handling represents a coining attribute that should be kept in mind.

9.3 Summary

We have seen various trends that might influence upcoming web architectures. The HTTP protocol will be eventually replaced by a new web protocol with better performance characteristics and several new features, as various alternative protocols already demonstrate.

The architecture of conventional RDBMS has been raised to question and alternative database systems will appear that get rid of legacy internals. In-memory database systems and event-sourced persistence represent two other and increasingly popular concepts for storage. When the latencies between nodes in a distributed system decrease, new remote operations and distributed computing patterns may become available.

Programming language trends are influenced by new web programming concepts and multi-paradigm language designs. The rise of virtual machines facilitates the formation of new programming languages.

For concurrent and distributed programming, functional and especially declarative programming languages are increasingly gaining attention. Some upcoming language concepts embrace the notions of nondeterminism, dataflow concurrency or logic programming. Others urge the developers to focus on associative, commutative and idempotent application operations in order to tame distributed programming.

10 Conclusion

We claim that concurrency is crucial for scalability, which in turn is inherently critical for large-scale architectures. The growing prevalence of web-based applications thus requires both scalable architectures and appropriate concepts for concurrent programming. Although web-based applications have always faced inherent parallelism, the concurrency implications for architectures and implementations are gradually changing.

The scalability of connection handling is not limited to increasing numbers of connections. For web real-time applications, web architectures are frequently confronted with very strict latency requirements. Interaction and notification mechanisms also demand the ability to handle huge numbers of mostly idle connections, in order to support server-side message pushing over HTTP. Also, mobile web applications have a bearing on connection performance and slow down the throughput of web servers. Similar requirements emerge for the application logic. Interactive web applications demand communication and coordination between multiple requests inside the business logic. In order to provide low latency responses, the application logic must utilize hardware resources as efficiently as possible. This yields highly concurrent execution environments for the business logic of web applications. Concurrency and scalability also challenge the persistence layer of a web architecture. The persistence layer must not only scale to very large data volumes, it must also handle increasingly concurrent read and write operations from the application layer. Large-scale web applications make the usage of distributed database systems inevitable. However, distributed database systems further increase the degree of complexity.

This thesis focused on devising an architectural model for scalable web architectures and then providing separate concurrency analyses of three main components: web servers, application servers and storage backends. Horizontal scalability and high availability have been the main requirements for the architectural design. We rejected a monolithic architecture due to complexity and scalability issues and campaigned for a structured, loosely coupled approach. Hence, the architecture is separated into functionally distinct components. Each of the components can be scaled separately and independently of other components. The components include load balancers, reverse caches, web servers, application servers, caching services, storage backends, background worker services, and an integration component for external services. For the most part, components are coupled using a message-based middleware.

We then provided a more detailed analysis of the concurrency internals for web servers, application servers and storage backends. We determined that massive I/O parallelism is the main challenge for web servers. We validated thread-based, event-driven and combined architectures for highly concurrent web servers. Next, we called attention to the duality argument of threads and events. We surmounted the general threads vs. events discussion and outlined the benefits of cooperative multithreading and asynchronous/non-blocking I/O operations for programming highly concurrent I/O-bound applications.

For the implementation of concurrent application logic, we assessed several programming concepts for concurrency from a generic viewpoint. The most common form of concurrent programming, based on threads, locks and shared state, is difficult and error-prone due to various reasons. Its usage should be limited to components where it is essential and inevitable. This includes low-level architecture components and the foundations for high-level concurrency libraries. For the actual application logic, higher concurrency abstractions are more advisable. Software transactional memory isolates operations on shared states similar to database systems. Hence, it allows lock-free programming and mitigates many problems of traditional multithreading. The actor model represents an entirely different approach that isolates the mutability of state. Actors are separate, single-threaded entities that communicate via immutable, asynchronous and guaranteed messaging. They encapsulate state and provide a programming model that embraces message-based distributed computing. Single-threaded event-driven application components are similar to actors—although they don't share the same architectural mind-set. Lesser known approaches include synchronous message passing and dataflow concepts.

Storage backends are mainly challenged by the CAP theorem. It disallows guaranteeing consistency and availability at the same time, when partitions must be tolerated at the same time. As a result, applications can either adhere to the conventional ACID properties, but must accept temporary unavailabilities—or they choose eventual consistency and make the application logic resilient to partially stale data. We illustrated that distributed database systems are based on relational or non-relational concepts and incorporate mechanisms from the database community and the distributed systems community.

We examined the relations between concurrency, scalability and distributed systems in general and outlined the underlying coherencies. We also provided some food for thought—for instance, which upcoming trends might influence future web architectures, and how distributed and concurrent programming can be prospectively handled.

Unfortunately, there is neither a silver bullet for the design of scalable architectures, nor for concurrent programming. We have extensively described the more popular approaches. We also compiled a list of recommendations for the design of concurrent and scalable web architectures of the near future. However, concurrency and scalability still remain taxing challenges for distributed systems in general and raise interesting research questions.

Bibliography

[Aba10] Abadi, Daniel: Problems with CAP, and Yahoo's little known NoSQL system, Blog Post: http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html (2010)

[Abb09] Abbott, Martin L. and Fisher, Michael T.: *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*, Addison-Wesley Professional (2009)

[Abb11] Abbott, Martin L. and Fisher, Michael T.: *Scalability Rules: 50 Principles for Scaling Web Sites*, Addison-Wesley Professional (2011)

[Ady02] Adya, Atul; Howell, Jon; Theimer, Marvin; Bolosky, William J. and Douceur, John R.: Cooperative Task Management Without Manual Stack Management, in: *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, USENIX Association, Berkeley, CA, USA, pp. 289–302

[Aga07] Agarwal, Aditya; Slee, Mark and Kwiatkowski, Marc: Thrift: Scalable Cross-Language Services Implementation, Tech. Rep., Facebook (2007)

[Agh90] Agha, Gul: Concurrent object-oriented programming. *Commun. ACM* (1990), vol. 33:pp. 125–141

[All08] Allspaw, John: *The Art of Capacity Planning: Scaling Web Resources*, O'Reilly Media (2008)

[All10] Allspaw, John and Robbins, Jesse: *Web Operations: Keeping the Data On Time*, O'Reilly Media (2010)

[Alv11] Alvaro, Peter; Conway, Neil; Hellerstein, Joe and Marczak, William R.: Consistency Analysis in Bloom: a CALM and Collected Approach, in: *CIDR*, pp. 249–260

[AMQ11] AMQP Working Group: AMQP Specification v1.0, Tech. Rep., Organization for the Advancement of Structured Information Standards (2011)

[And10] Anderson, J. Chris; Lehnardt, Jan and Slater, Noah: *CouchDB: The Definitive Guide: Time to Relax (Animal Guide)*, O'Reilly Media (2010)

[Arm07] Armstrong, Joe: *Programming Erlang: Software for a Concurrent World*, Pragmatic Bookshelf (2007)

[Bak77] Baker, Henry G., Jr. and Hewitt, Carl: The Incremental Garbage Collection of Processes, Tech. Rep., Cambridge, MA, USA (1977)

[Bel09] Belshe, Mike: SPDY: An experimental protocol for a faster web, Tech. Rep., Google Inc. (2009)

[Ber81] Bernstein, Philip A. and Goodman, Nathan: Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* (1981), vol. 13(2):pp. 185–221

[BL01] Berners-Lee, Tim; Hendler, James and Lassila, Ora: The Semantic Web. *Scientific American* (2001), vol. 284(5):pp. 34–43

[BL05] Berners-Lee, T.; Fielding, R. and Masinter, L.: Uniform Resource Identifier (URI): Generic Syntax, RFC 3986 (Standard) (2005)

[Blo08] Bloch, Joshua: *Effective Java (2nd Edition)*, Addison-Wesley (2008)

[Bra08] Bray, Tim; Paoli, Jean; Maler, Eve; Yergeau, François and Sperberg-McQueen, C. M.: Extensible Markup Language (XML) 1.0 (Fifth Edition), W3C recommendation, W3C (2008), http://www.w3.org/TR/2008/REC-xml-20081126/

[Bre00] Brewer, Eric A.: Towards robust distributed systems (abstract), in: *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, ACM, New York, NY, USA, pp. 7–

[Bro87] Brooks, Frederick P., Jr.: No Silver Bullet Essence and Accidents of Software Engineering. *Computer* (1987), vol. 20(4):pp. 10–19

[Can08] Cantrill, Bryan and Bonwick, Jeff: Real-World Concurrency. *Queue* (2008), vol. 6:pp. 16–25

[Cas08] Cascaval, Calin; Blundell, Colin; Michael, Maged; Cain, Harold W.; Wu, Peng; Chiras, Stefanie and Chatterjee, Siddhartha: Software Transactional Memory: Why Is It Only a Research Toy? *Queue* (2008), vol. 6:pp. 46–58

[Cha06] Chang, Fay; Dean, Jeffrey; Ghemawat, Sanjay; Hsieh, Wilson C.; Wallach, Deborah A.; Burrows, Mike; Chandra, Tushar; Fikes, Andrew and Gruber, Robert E.: Bigtable: A distributed storage system for structured data, in: *In Proceedings Of The 7Th Conference On Usenix Symposium On Operating Systems Design And Implementation - Volume 7*, pp. 205–218

[Cle04] Clement, Luc; Hately, Andrew; von Riegen, Claus and Rogers, Tony: UDDI Spec Technical Committee Draft 3.0.2, Oasis committee draft, OASIS (2004)

[Cod70] Codd, E. F.: A relational model of data for large shared data banks. *Commun. ACM* (1970), vol. 13(6):pp. 377–387

[Con63] Conway, Melvin E.: Design of a separable transition-diagram compiler. *Commun. ACM* (1963), vol. 6(7):pp. 396–408

[Cre09] Creeger, Mache: Cloud Computing: An Overview. *Queue* (2009), vol. 7:pp. 2:3–2:4

[Cro06] Crockford, D.: The application/json Media Type for JavaScript Object Notation (JSON), RFC 4627 (Informational) (2006)

[Cro08] Crockford, Douglas: *JavaScript: The Good Parts*, Yahoo Press (2008)

[Dah09] Dahan, Udi: Clarified CQRS, Tech. Rep., udidahan.com (2009)

[Dea08] Dean, Jeffrey and Ghemawat, Sanjay: MapReduce: simplified data processing on large clusters. *Commun. ACM* (2008), vol. 51:pp. 107–113

[DeC07] DeCandia, Giuseppe; Hastorun, Deniz; Jampani, Madan; Kakulapati, Gunavardhan; Lakshman, Avinash; Pilchin, Alex; Sivasubramanian, Swaminathan; Vosshall, Peter and Vogels, Werner: Dynamo: amazon's highly available key-value store, in: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, ACM, New York, NY, USA, pp. 205–220

[Dij65] Dijkstra, Edsger Wybe: Cooperating Sequential Processes, Technical Report EWD-123, Tech. Rep., University of Texas at Austin (1965)

[Dol05] Dollimore, Jean; Kindberg, Tim and Coulouris, George: *Distributed Systems: Concepts and Design (4th Edition)*, Addison Wesley (2005)

[Dra11] Dragojević, Aleksandar; Felber, Pascal; Gramoli, Vincent and Guerraoui, Rachid: Why STM can be more than a research toy. *Commun. ACM* (2011), vol. 54:pp. 70–77

[ECM99] ECMA International: Standard ECMA-262, Tech. Rep., ECMA International (1999)

[Eis99] Eisenberg, Andrew and Melton, Jim: SQL: 1999, formerly known as SQL3. *SIGMOD Rec.* (1999), vol. 28(1):pp. 131–138

[Esw76] Eswaran, K. P.; Gray, J. N.; Lorie, R. A. and Traiger, I. L.: The notions of consistency and predicate locks in a database system. *Commun. ACM* (1976), vol. 19(11):pp. 624–633

[Fet11] Fette, I. and Melnikov, A.: The WebSocket Protocol, RFC 6455 (Informational) (2011)

[Fie99] Fielding, R.; Gettys, J.; Mogul, J.; Frystyk, H.; Masinter, L.; Leach, P. and Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1, RFC 2616 (Draft Standard) (1999), updated by RFCs 2817, 5785, 6266

[Fie00] Fielding, Roy Thomas: *Architectural styles and the design of network-based software architectures*, Ph.D. thesis, University of California, Irvine (2000), aAI9980887

[Fly72] Flynn, Michael J.: Some computer organizations and their effectiveness. *IEEE Trans. Comput.* (1972), vol. 21(9):pp. 948–960

[For12] Foresti, A.; Singhal, S.; Mazahir, O.; Nielsen, H.; Raymor, B.; Rao, R. and Montenegro, G.: HTTP Speed+Mobility, Tech. Rep., Microsoft (2012)

[Fow02] Fowler, Martin: *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional (2002)

[Fow05] Fowler, Martin: Event Sourcing, Tech. Rep., ThoughtWorks (2005)

[Fri76] Friedman, Daniel and Wise, David: The Impact of Applicative Programming on Multiprocessing, in: *International Conference on Parallel Processing*

[Fri99] Frigo, Matteo; Leiserson, Charles E.; Prokop, Harald and Ramachandran, Sridhar: Cache-Oblivious Algorithms, in: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, IEEE Computer Society, Washington, DC, USA, pp. 285–

[fS86] for Standardization, International Organization: *ISO 8879:1986: Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*, International Organization for Standardization (1986)

[Gho10] Ghosh, Debasish: *DSLs in Action*, Manning Publications (2010)

[Gho11] Ghosh, Debasish; Sheehy, Justin; Thorup, Kresten and Vinoski, Steve: Programming language impact on the development of distributed systems. *Journal of Internet Services and Applications* (2011), vol. Issue 2 / 2011:pp. 1–8, 10.1007/s13174-011-0042-y

[Gif79] Gifford, David K.: Weighted voting for replicated data, in: *Proceedings of the seventh ACM symposium on Operating systems principles*, SOSP '79, ACM, New York, NY, USA, pp. 150–162

[Gil02] Gilbert, Seth and Lynch, Nancy: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* (2002), vol. 33:pp. 51–59

[Gm92] Garcia-molina, Hector and Salem, Kenneth: Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering* (1992), vol. 4:pp. 509–516

[Goe06] Goetz, Brian; Peierls, Tim; Bloch, Joshua; Bowbeer, Joseph; Holmes, David and Lea, Doug: *Java Concurrency in Practice*, Addison-Wesley Professional (2006)

[Goe10] Goetz, Brian: JSR 335: Lambda Expressions for the Java(TM) Programming Language, Tech. Rep., Oracle Inc. (2010)

[Gos12] Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad and Buckley, Alex: The Java Language Specification, Java SE 7 Edition, Tech. Rep., Oracle Inc. (2012)

[Gro07] Grossman, Dan: The transactional memory / garbage collection analogy, in: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, ACM, New York, NY, USA, pp. 695–706

[Gud07] Gudgin, Martin; Hadley, Marc; Mendelsohn, Noah; Lafon, Yves; Moreau, Jean-Jacques; Karmarkar, Anish and Nielsen, Henrik Frystyk: SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), W3C recommendation, W3C (2007), http://www.w3.org/TR/2007/REC-soap12-part1-20070427/

[Gue07] Guerraoui, Rachid: A Smooth Concurrency Revolution with Free Objects. *IEEE Internet Computing* (2007), vol. 11:pp. 82–85

[Gus05] Gustafsson, Andreas: Threads without the Pain. *Queue* (2005), vol. 3:pp. 34–41

[Hae83] Haerder, Theo and Reuter, Andreas: Principles of transaction-oriented database recovery. *ACM Comput. Surv.* (1983), vol. 15(4):pp. 287–317

[Hal06] Haller, Philipp and Odersky, Martin: Event-Based Programming without Inversion of Control, in: David E. Lightfoot and Clemens A. Szyperski (Editors) *Modular Programming Languages*, Lecture Notes in Computer Science, pp. 4–22

[Hal08] Haller, Philipp and Odersky, Martin: Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science* (2008)

[Har08] Harris, Tim; Marlow, Simon; Jones, Simon Peyton and Herlihy, Maurice: Composable memory transactions. *Commun. ACM* (2008), vol. 51:pp. 91–100

[Hel07] Hellerstein, Joseph M.; Stonebraker, Michael and Hamilton, James: *Architecture of a Database System*, Now Publishers Inc., Hanover, MA, USA (2007)

[Hel09] Helland, Pat and Campbell, David: Building on Quicksand, in: *CIDR*

[Hel10] Hellerstein, Joseph M.: The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.* (2010), vol. 39:pp. 5–19

[Hel12] Helland, Pat: Idempotence Is Not a Medical Condition. *Queue* (2012), vol. 10(4):pp. 30:30–30:46

[Her93] Herlihy, Maurice and Moss, J. Eliot B.: Transactional memory: architectural support for lock-free data structures, in: *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, ACM, New York, NY, USA, pp. 289–300

[Hew73] Hewitt, Carl; Bishop, Peter and Steiger, Richard: A universal modular ACTOR formalism for artificial intelligence, in: *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 235–245

[Hic08] Hickey, Rich: The Clojure programming language, in: *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, ACM, New York, NY, USA, pp. 1:1–1:1

[Hic09a] Hickson, Ian: The Web Sockets API, W3C working draft, W3C (2009), http://www.w3.org/TR/2009/WD-websockets-20091222/

[Hic09b] Hickson, Ian: Web Storage, Last call WD, W3C (2009), http://www.w3.org/TR/2009/WD-webstorage-20091222/

[Hic09c] Hickson, Ian: Web Workers, Last call WD, W3C (2009), http://www.w3.org/TR/2009/WD-workers-20091222/

[Hoa74] Hoare, C. A. R.: Monitors: an operating system structuring concept. *Commun. ACM* (1974), vol. 17(10):pp. 549–557

[Hoa78] Hoare, C. A. R.: Communicating sequential processes. *Commun. ACM* (1978), vol. 21(8):pp. 666–677

[Hoh03] Hohpe, Gregor and Woolf, Bobby: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley Professional (2003)

[Hoh06] Hohpe, Gregor: Programming Without a Call Stack – Event-driven Architectures, Tech. Rep., eaipatterns.com (2006)

[Hya09] Hyatt, David and Hickson, Ian: HTML 5, W3C working draft, W3C (2009), http://www.w3.org/TR/2009/WD-html5-20090825/

[Ire09] Ireland, Christopher; Bowers, David; Newton, Michael and Waugh, Kevin: A Classification of Object-Relational Impedance Mismatch, in: *Proceedings of the 2009 First International Conference on Advances in Databases, Knowledge, and Data Applications*, DBKDA '09, IEEE Computer Society, Washington, DC, USA, pp. 36–43

[Jac99] Jacobs, Ian; Raggett, David and Hors, Arnaud Le: HTML 4.01 Specification, W3C recommendation, W3C (1999), http://www.w3.org/TR/1999/REC-html401-19991224

[Kar97] Karger, David; Lehman, Eric; Leighton, Tom; Panigrahy, Rina; Levine, Matthew and Lewin, Daniel: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web, in: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, ACM, New York, NY, USA, pp. 654–663

[Keg06] Kegel, Dan: The C10K problem, Tech. Rep., Kegel.com (2006)

[Ken94] Kendall, Samuel C.; Waldo, Jim; Wollrath, Ann and Wyant, Geoff: A Note on Distributed Computing, Tech. Rep., Sun Microsystems Laboratories, Mountain View, CA, USA (1994)

[Kni86] Knight, Tom: An architecture for mostly functional languages, in: *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, ACM, New York, NY, USA, pp. 105–112

[Lam78] Lamport, Leslie: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* (1978), vol. 21(7):pp. 558–565

[Lam79a] Lampson, B. and Sturgis, H.: Crash recovery in a distributed storage system, Tech. Rep., Xerox Palo Alto Research Center, Palo Alto, CA (1979)

[Lam79b] Lampson, Butler W. and Redell, David D.: Experience with processes and monitors in Mesa (Summary), in: *Proceedings of the seventh ACM symposium on Operating systems principles*, SOSP '79, ACM, New York, NY, USA, pp. 43–44

[Lam98] Lamport, Leslie: The part-time parliament. *ACM Trans. Comput. Syst.* (1998), vol. 16(2):pp. 133–169

[Lar08] Larson, Jim: Erlang for Concurrent Programming. *Queue* (2008), vol. 6:pp. 18–23

[Lau79] Lauer, Hugh C. and Needham, Roger M.: On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.* (1979), vol. 13:pp. 3–19

[Lea00] Lea, Doug: A Java fork/join framework, in: *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, ACM, New York, NY, USA, pp. 36–43

[Lee06] Lee, Edward A.: The Problem with Threads. *Computer* (2006), vol. 39:pp. 33–42

[Les09] Lesani, Mohsen; Odersky, Martin and Guerraoui, Rachid: Transactors: Unifying Transactions and Actors, Tech. Rep. (2009)

[Les11] Lesani, Mohsen and Palsberg, Jens: Communicating memory transactions, in: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, ACM, New York, NY, USA, pp. 157–168

[Li07] Li, Peng and Zdancewic, Steve: Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives, in: *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, ACM, New York, NY, USA, pp. 189–199

[Lis88] Liskov, B. and Shrira, L.: Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.* (1988), vol. 23(7):pp. 260–267

[Liu07] Liu, Canyang Kevin and Booth, David: Web Services Description Language (WSDL) Version 2.0 Part 0: Primer, W3C recommendation, W3C (2007), http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626

[Mar12] Marr, Stefan and D'Hondt, Theo: Identifying A Unifying Mechanism for the Implementation of Concurrency Abstractions on Multi-Language Virtual Machines, in: *Objects, Models, Components, Patterns, 50th International Conference, TOOLS 2012*, Springer, (to appear)

[Mas98] Masinter, L.: The "data" URL scheme, RFC 2397 (Proposed Standard) (1998)

[Mat09] Mattson, Robert L. R. and Ghosh, Somnath: HTTP-MPLEX: An enhanced hypertext transfer protocol and its performance evaluation. *J. Netw. Comput. Appl.* (2009), vol. 32(4):pp. 925–939

[McG11] McGranaghan, Mark: ClojureScript: Functional Programming for JavaScript Platforms. *IEEE Internet Computing* (2011), vol. 15(6):pp. 97–102

[Mey01] Meyer, Eric A. and Bos, Bert: CSS3 introduction, W3C working draft, W3C (2001), http://www.w3.org/TR/2001/WD-css3-roadmap-20010523/

[Moi11] Moiz, Salman Abdul; P., Sailaja; G., Venkataswamy and Pal, Supriya N.: Article: Database Replication: A Survey of Open Source and Commercial Tools. *International Journal of Computer Applications* (2011), vol. 13(6):pp. 1–8, published by Foundation of Computer Science

[Mol03] Molnar, Ingo: The Native POSIX Thread Library for Linux, Tech. Rep., Tech. Rep., RedHat, Inc (2003)

[Mos06] Moseley, Ben and Marks, Peter: Out of the Tar Pit, Tech. Rep. (2006)

[Mur10] Murray, Derek G. and Hand, Steven: Scripting the cloud with skywriting, in: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, USENIX Association, Berkeley, CA, USA, pp. 12–12

[Mur11] Murray, Derek G.; Schwarzkopf, Malte; Smowton, Christopher; Smith, Steven; Madhavapeddy, Anil and Hand, Steven: CIEL: a universal execution engine for distributed data-flow computing, in: *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, USENIX Association, Berkeley, CA, USA, pp. 9–9

[Not11] Nottingham, Mark: On HTTP Load Testing, Blog Post: http://www.mnot.net/blog/2011/05/18/http_benchmark_rules (2011)

[Not12] Nottingham, Mark: What's Next for HTTP, Blog Post: http://www.mnot.net/blog/2012/03/31/whats_next_for_http (2012)

[Oka96] Okasaki, Chris: *Purely Functional Data Structures*, Ph.D. thesis, Carnegie Mellon University (1996)

[Ous96] Ousterhout, John: Why Threads are a Bad Idea (for most purposes), in: *USENIX Winter Technical Conference*

[Pai99] Pai, Vivek S.; Druschel, Peter and Zwaenepoel, Willy: Flash: an efficient and portable web server, in: *Proceedings of the annual conference on USENIX Annual Technical Conference*, USENIX Association, Berkeley, CA, USA, pp. 15–15

[Par07] Pariag, David; Brecht, Tim; Harji, Ashif; Buhr, Peter; Shukla, Amol and Cheriton, David R.: Comparing the performance of web server architectures, in: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, ACM, New York, NY, USA, pp. 231–243

[Pri08] Pritchett, Dan: BASE: An Acid Alternative. *Queue* (2008), vol. 6(3):pp. 48–55

[Pya97] Pyarali, Irfan; Harrison, Tim; Schmidt, Douglas C. and Jordan, Thomas D.: Proactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events, Tech. Rep., Washington University (1997)

[Ran78] Randell, B.; Lee, P. and Treleaven, P. C.: Reliability Issues in Computing System Design. *ACM Comput. Surv.* (1978), vol. 10(2):pp. 123–165

[Ran10] Randles, Martin; Lamb, David and Taleb-Bendiab, A.: A Comparative Study into Distributed Load Balancing Algorithms for Cloud Computing, in: *Proceedings of the 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, WAINA '10, IEEE Computer Society, Washington, DC, USA, pp. 551–556

[Ree78] Reed, D. P.: *Naming And Synchronization In A Decentralized Computer System*, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (1978)

[RGO06] Rotem-Gal-Oz, Arnon; Gosling, James and Deutsch, L. Peter: Fallacies of Distributed Computing Explained, Tech. Rep., Sun Microsystems (2006)

[Rob04] Robinson, D. and Coar, K.: The Common Gateway Interface (CGI) Version 1.1, RFC 3875 (Informational) (2004)

[Ros11] Rose, John: SR 292: Supporting Dynamically Typed Languages on the Java(TM) Platform, Tech. Rep., Oracle America, Inc. (2011)

[Roy04] Roy, Peter Van and Haridi, Seif: *Concepts, Techniques, and Models of Computer Programming*, The MIT Press (2004)

[Rum11] Rumble, Stephen M.; Ongaro, Diego; Stutsman, Ryan; Rosenblum, Mendel and Ousterhout, John K.: It's time for low latency, in: *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, USENIX Association, Berkeley, CA, USA, pp. 11–11

[Rys11] Rys, Michael: Scalable SQL. *Commun. ACM* (2011), vol. 54(6):pp. 48–53

[Sch95] Schmidt, Douglas C.: *Reactor: an object behavioral pattern for concurrent event demultiplexing and event handler dispatching*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1995), pp. 529–545

[Sch06] Schlossnagle, Theo: *Scalable Internet Architectures*, Sams (2006)

[Sch08] Schwartz, Baron; Zaitsev, Peter; Tkachenko, Vadim; D., Jeremy Zawodny; Lentz, Arjen and Balling, Derek J.: *High Performance MySQL: Optimization, Backups, Replication, and More*, O'Reilly Media (2008)

[Seb05] Sebesta, Robert W.: *Concepts of Programming Languages (7th Edition)*, Addison Wesley (2005)

[Sha00] Shachor, Gal and Milstein, Dan: The Apache Tomcat Connector - AJP Protocol Reference, Tech. Rep., Apache Software Foundation (2000)

[Ske82] Skeen, Dale: A Quorum-Based Commit Protocol, Tech. Rep., Cornell University, Ithaca, NY, USA (1982)

[Ske83] Skeen, D. and Stonebraker, M.: A Formal Model of Crash Recovery in a Distributed System. *IEEE Trans. Softw. Eng.* (1983), vol. 9(3):pp. 219–228

[Ste03] Stevens, W. Richard; Fenner, Bill and Rudoff, Andrew M.: *Unix Network Programming, Volume 1: The Sockets Networking API (3rd Edition)*, Addison-Wesley Professional (2003)

[Sto07] Stonebraker, Michael; Madden, Samuel; Abadi, Daniel J.; Harizopoulos, Stavros; Hachem, Nabil and Helland, Pat: The end of an architectural era: (it's time for a complete rewrite), in: *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, VLDB Endowment, pp. 1150–1160

[Sto10] Stonebraker, Michael: Errors in Database Systems, Eventual Consistency, and the CAP Theorem, Web Article: http://bit.ly/cCDWDS (2010)

[Sut05] Sutter, Herb and Larus, James: Software and the Concurrency Revolution. *Queue* (2005), vol. 3:pp. 54–62

[Tan06] Tanenbaum, Andrew S. and Steen, Maarten Van: *Distributed Systems: Principles and Paradigms (2nd Edition)*, Prentice Hall (2006)

[Tea12] Team, The Dart: Dart Programming Language Specification, Tech. Rep., Google Inc. (2012)

[Tha10] Thalinger, Christian and Rose, John: Optimizing invokedynamic, in: *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, ACM, New York, NY, USA, pp. 1–9

[Tho11] Thompson, Martin; Farley, Dave; Barker, Michael; Gee, Patricia and Stewart, Andrew: Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads, Tech. Rep., LMAX (2011)

[Til10] Tilkov, Stefan and Vinoski, Steve: Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing* (2010), vol. 14:pp. 80–83

[Ung10] Ungar, David and Adams, Sam S.: Harnessing emergence for manycore programming: early experience integrating ensembles, adverbs, and object-based inheritance, in: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, ACM, New York, NY, USA, pp. 19–26

[vB03a] von Behren, Rob; Condit, Jeremy and Brewer, Eric: Why events are a bad idea (for high-concurrency servers), in: *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, USENIX Association, Berkeley, CA, USA, pp. 4–4

[vB03b] von Behren, Rob; Condit, Jeremy; Zhou, Feng; Necula, George C. and Brewer, Eric: Capriccio: scalable threads for internet services, in: *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, ACM, New York, NY, USA, pp. 268–281

[Vin07] Vinoski, Steve: Concurrency with Erlang. *IEEE Internet Computing* (2007), vol. 11:pp. 90–93

[Vin08] Vinoski, Steve: Convenience Over Correctness. *IEEE Internet Computing* (2008), vol. 12:pp. 89–92

[Vin09] Vinoski, Steve: Welcome to "The Functional Web". *IEEE Internet Computing* (2009), vol. 13:pp. 104–103

[Vog08] Vogels, Werner: Eventually Consistent. *Queue* (2008), vol. 6(6):pp. 14–19

[Wel01] Welsh, Matt; Culler, David and Brewer, Eric: SEDA: an architecture for well-conditioned, scalable internet services, in: *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, ACM, New York, NY, USA, pp. 230–243

[Wel10] Welsh, Matt: A Retrospective on SEDA, Blog Post: http://matt-welsh.blogspot.com/2010/07/retrospective-on-seda.html (2010)

[Win99] Winer, Dave: XML-RPC Specification, Tech. Rep., UserLand Software, Inc. (1999)

[Zah10] Zaharia, Matei; Chowdhury, Mosharaf; Franklin, Michael J.; Shenker, Scott and Stoica, Ion: Spark: cluster computing with working sets, in: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, USENIX Association, Berkeley, CA, USA, pp. 10–10

[Zel03] Zeldovich, Nickolai; Yip, Er; Dabek, Frank; Morris, Robert T.; Mazières, David and Kaashoek, Frans: Multiprocessor support for event-driven programs, in: *In Proceedings of the 2003 USENIX Annual Technical Conference (USENIX '03*, pp. 239–252

[Zhu11] Zhu, Wenbo: Implications of Full-Duplex HTTP, Tech. Rep., Google, Inc. (2011)