
Scale The Realtime Web

Bachelor Thesis

Sven Pfeiderer

Hochschule der Medien, Stuttgart
Studiengang Medieninformatik

by

Sven Pfeiderer
(sven@roothausen.de)

Primary thesis advisor: Prof. Walter Kriha

Secondary thesis advisor: Marc Seeger

01.03.2011

Abstract

Over the last few years we watched a paradigm shift from static content provided by a few entities to dynamic, user generated content. This content is no longer generated by only a few chosen ones. It is now generated by virtually everybody. Users no longer look at websites every few hours but want to know instantly if there are any news. They also expect, that content they create is visible to their peers instantly.

To fulfill this need for instant updates, more and more websites added dynamic content loading. They changed from websites which provided content to web applications with desktop like features. The next logical step was the transition to “realtime web applications” which allow much faster updates. This generation of web applications will not only enable developers to create realtime collaboration and communication applications but also provide a basis of fast browser based games.

This thesis will discuss the current state of realtime web applications, the need for high performance application servers and design patterns to handle a huge amount of clients with persistent connections to the server.

Acknowledgements

This thesis has not been written in a void so I would like to use this opportunity to thank all people involved in its creation. Especially Walter Kriha for sparking my interest for distributed systems, the inspiration and the support. I also like to thank my good friend Felix 'makefu' Richter who didn't get tired of me asking questions about lowlevel UNIX programming and Marc Seeger who was always open for discussions on interesting topics.

I also contacted a number of IT professionals and asked them for their opinion regarding the topic of my thesis. Last but not least, I would like to thank David Pollack, Roberto Ostinelli and Ilya Grigorik for their valuable contributions.

Statement Of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices.

Sven Pfeiderer, March 2011

Contents

Table of Contents	8
List of Figures	9
1 Introduction	10
1.1 The Rise Of Web Applications	10
1.1.1 Fundamentals	10
1.1.2 Popularity	10
1.1.3 Technical Improvements	11
1.2 Example Use Case: Realtime Chat	12
1.3 Motivation	14
1.4 Structure Of This Document	14
2 Client Side	16
2.1 Ajax	16
2.1.1 What Is Ajax?	16
2.1.2 Example Using Ajax	17
2.1.3 Ajax Drawbacks	17
2.2 Comet	19
2.2.1 What Is Comet?	19
2.2.2 Example Using Comet	19
2.2.3 Comet Drawbacks	20
2.3 WebSockets	21
2.3.1 What Are WebSockets?	21
2.3.2 Example Using WebSockets	21
2.3.3 Drawbacks	22
2.4 Put Them All Together	23
3 Problems With Realtime Web Applications	24
3.1 Blocking Calls	24

3.2	The End Of A Simple Request/Response Model	25
3.3	Statefulness	25
3.4	Long Living TCP Connections	26
3.5	Bidirectional, Asynchronous Communication	26
4	Server Side	28
4.1	Processes	28
4.1.1	Process Basics	28
4.1.2	Handle incoming connections with one process	29
4.1.3	Handle incoming connections with many processes	30
4.1.4	Process Optimizations	32
4.1.5	Constraints Of Processes	33
4.1.6	Conclusion	34
4.2	Threads	35
4.2.1	Threading Basics	35
4.2.2	Native Threads vs. Green Threads	35
4.2.3	N:M Threading vs. 1:N Threading vs. 1:1 Threading	36
4.2.4	Example Using Threads	37
4.2.5	Constraints Of Threads	37
4.2.6	Conclusion	40
4.3	Event Loop	42
4.3.1	Event Loop Basics	42
4.3.2	Reactor Pattern	44
4.3.3	Proactor Pattern	44
4.3.4	Underlying UNIX APIs	45
4.3.5	Example Using An Event Loop	47
4.3.6	Constraints of an Event Loop	48
4.3.7	Conclusion	50
4.4	Actor Model	52
4.4.1	Actor Basics	52
4.4.2	Fault Tolerance	54
4.4.3	Remote Actors	54
4.4.4	Actors Using “Green Processes” – Erlang Style	54
4.4.5	Actors On A Threadpool	56
4.4.6	Example Using Actors	56
4.4.7	Constraints Of The Actor Model	58

4.4.8	Conclusion	59
4.5	Coroutines	60
4.5.1	Coroutine Basics	60
4.5.2	Coroutine API	61
4.5.3	Coroutines And Generators	62
4.5.4	Coroutines And Cooperative Concurrency	63
4.5.5	Example Using Coroutines	63
4.5.6	Constraints Of Coroutines	65
4.5.7	Conclusion	67
4.6	Hybrid Systems	68
4.6.1	Event-Loop with Threads	68
4.6.2	Event-Loops On Multiple Processes	68
4.6.3	Event-Loop And Co-Routines	68
4.6.4	Actors On A Threadpool	69
4.6.5	SEDA	69
5	Frameworks	70
5.1	Lift	70
5.2	EventMachine	70
5.3	Twisted	71
5.4	Aleph	71
5.5	Misultin	71
5.6	Node.js	71
6	Conclusions	73
6.1	There Are No Silver Bullets	73
6.2	Differentiate Between Concurrency And Parallelism	73
6.2.1	Concurrency And Parallelism	73
6.2.2	Concurrency And Parallelism In Realtime Web Applications	74
6.3	The Future Will Be Async	74
6.4	Use The Right Tool For The Job	75
6.5	A Need For Better Abstractions To Model Concurrency	75
	Bibliography	80
7	Appendix	81
7.1	David Pollack, Creator Of the Lift Web Framework	81

7.2	Ilya Grigorik, Technology Blogger and Founder Of Postrank	82
7.3	Roberto Ostinelli, Creator Of The Misultin Framework	83

List of Figures

1.1	Screenshot of the user interface of the example application	13
4.1	From left to right: 1:N Threading, N:M Threading, 1:1 Threading	36
4.2	A deadlock caused by two threads	39
4.3	Comparison of preemptive scheduling with cooperative scheduling	60

1 Introduction

Web applications are very popular these days. This popularity and the need for high performance software is the main motivation behind this thesis. This chapter will show some reasons for this development as well as explain the example application used as an example throughout this thesis.

1.1 The Rise Of Web Applications

1.1.1 Fundamentals

A web based application, or web application, is an application which is accessed over a network and executed from a web browser. Most of these applications are accessible over the Internet and are based on web technologies. The only thing a user needs to run this software is access to the Internet and a web browser capable to render HTML and CSS and able to execute JavaScript.

1.1.2 Popularity

Web applications are very popular nowadays, but they might get even more popular in the near future. There are some good reasons for users to choose web applications over their desktop competitors:

- The ubiquity of the World Wide Web
- Web applications run on a lot of platforms and devices
- There is no need to install anything except a web browser
- Data is stored on servers and can be accessed from virtually everywhere

But in an operations perspective there are even more benefits:

- Programmers can choose the technology they want to create the application
- Updates can be deployed immediately
- Data needed to debug and profile the application is directly accessible
- There is no need to worry about operating system, device drivers or drive space on the client side

Or like Paul Graham put it in his book “Hackers and Painters” [Graham, 2004]

When we look back on the desktop software era, I think we’ll marvel at the inconveniences people put up with, just as we marvel now at what early car owners put up with. For the first twenty or thirty years, you had to be a car expert to own a car. But cars were such a big win that lots of people who weren’t car experts wanted to have them as well.

1.1.3 Technical Improvements

High Performance JavaScript Runtimes Over the last couple of years JavaScript runtimes, like Google’s V8 [Google, 2010], improved a lot: Just in time compilers (JIT), better garbage collectors and optimizers increased the performance of JavaScript runtimes significantly. Process sandboxing and permission systems for extensions also lead to more reliable and more secure runtimes. JavaScript itself had a reputation as “toy language”. Due to the described changes and more mature development tools, the developers’ view of JavaScript changed. JavaScript is taken serious these days and used for client as well as server side applications.

HTML5 Not only JavaScript evolved in the last few years. There were also a lot of improvements regarding the markup of web applications: The new upcoming conglomerate of standards called HTML5 [Hickson and Hyatt, 2010] includes tags to embed video and audio content, allows to include inline vector graphics using SVG¹ and even has new tags to enable validation of form fields. In short: There is a lot of movement which will enable developers to create even more advanced applications.

¹Scalable Vector Graphics

CSS3 Another source of innovation is the language used to design web applications: CSS. To make applications more dynamic and easy to use, CSS3 [Eric A. Meyer, 2001] enables developers and designers to use declarative animations, create colored gradients and use a new syntax called “CSS-Selectors” to select elements.

1.2 Example Use Case: Realtime Chat

A chat application was chosen to be a simple enough real-world example for demonstrating the different technologies on the server as well as the client. It should fulfill the following requirements:

- Connect a user with all other users watching the same website
- Show a message from one user immediately to all other users
- Provide a responsive user interface

As seen in Figure 1.1, the user interface of this application is very basic. But nevertheless it shows some important features of realtime web applications: The application needs to be able to receive and send messages at the same time and also needs to be able to display new incoming messages with no noticeable delay.

To make a chat application useful, it must be able to support many concurrent users who are chatting with each other. For this reason the server should be able to handle lots of established connections, process their messages and deliver them with no noticeable delay.

In times of web applications which suddenly need to handle an immense lot of users once the application gains some popularity, it will most likely also be necessary to scale the application vertically and horizontally. Scaling vertically or “to scale up” means to be able to handle more users on one particular machine by adding more resources like memory or CPUs. Scaling horizontally, or “to scale out”, on the other side, means to distribute the load over a network of machines.



Figure 1.1: Screenshot of the user interface of the example application

1.3 Motivation

As stated before in Section 1.1, the Web is a very interesting and promising platform, for both application developers and users. It provides a lot of new possibilities and new upcoming web based technologies will help to increase this possibilities even further.

On the other side, the change from simple documents viewed in a web browser towards complex web applications with realtime features leads to considerable changes in application architecture and infrastructure. As applications grow bigger and more users are using them, there is a strong need for high performance application servers and development frameworks. Both, developers and users, have needs in terms of performance and usability which need to be addressed by the used technology.

To build high performance systems, developers need knowledge of mechanics used to built web- and application servers and need to understand the implications of the technologies they use. The main goal of the thesis is therefore to summarize and explain the current state of realtime web applications. Or as Aviv Ben-Yosef stated in his article “You Owe it to Yourself to be Old-School” [Ben-Yosef, 2011]:

I don't care if you're doing Rails and never need to see the outside of a pointer. There's nothing like having the holistic grasp of things to help you solve problems quickly, a-la Dirk Gently.

1.4 Structure Of This Document

This thesis is divided into six chapters:

Chapter 1 gives an introduction to the topic, explains the success of web applications and illustrates the example application used in this thesis.

Chapter 2 introduces client side techniques and standards used to build realtime web applications. It also shows code examples written in JavaScript which illustrate the usage of different techniques to realize more dynamic web applications.

Chapter 3 shows the major differences between “classical” web applications and realtime web applications and explains the need for high performance application servers.

Chapter 4 represents the core part of this thesis and explains the most common approaches to servers which need to support concurrent users. It shows the difference between processes, threads, actors, event loops and coroutines as well as some examples of hybrid systems. Most topics are also illustrated with code examples.

Chapter 5 shows six different web development frameworks and their approach to concurrency.

Chapter 6 wraps up the previous sections and elaborates the most important observations made by the author while writing this thesis.

2 Client Side

This chapter will explain client side technologies used to enable realtime web applications as well as their up- and downsides. The mentioned technologies will be explained with the help of some code examples using the example chat application described in section 1.2.

2.1 Ajax

2.1.1 What Is Ajax?

Ajax, a shorthand for Asynchronous JavaScript and XML, was first introduced as an acronym by Jesse James Garrett in his article “Ajax: A New Approach to Web Applications” [Garrett, 2005]. Garret was describing a paradigm shift in web application technology which enabled application developers to create more responsive web applications with desktop like features.

The term Ajax describes the use of several technologies like dynamic manipulation of the browser Document Object Model (DOM), XML and XSL for data exchange and data retrieval via asynchronous XMLHttpRequest. When put together, these technologies allow a retrieval of new data from the server, process it and display it without a need to reload the whole website. Even if XML is not the primary data exchange format today and was widely replaced by JSON¹ and pre-rendered HTML, Ajax is still used heavily to make web applications more interactive and responsive.

¹JavaScript Object Notation

2.1.2 Example Using Ajax

As shown in Listing 2.1, a chat application using Ajax needs to implement a polling mechanism which periodically asks the server for new chat messages. This mechanism calls an URL where new messages are provided by the server. It uses the *ajaxComplete()* function as a callback. It is called immediately when the request finishes. New messages are passed as a parameter to *ajaxComplete()*. These new messages need to be processed after retrieval. One second after the request completes, a new request will be sent.

To be also able to send new messages, the client needs also a mechanism to send them to the server. In the example code this is implemented with a click handler that will be called, when the “send”-button is clicked.

```
1 // callback for ajaxCall
2 function ajaxComplete(response) {
3   // process messages
4   addNewMessagesToChat(response);
5   // make new call after one second
6   setTimeout(1000, ajaxCall);
7 };
8
9 function ajaxCall() {
10  // get new messages from server, register callback
11  jQuery.get('/path/to/newmessages', ajaxComplete, 'json');
12 };
13
14 // register click handler to send new message
15 jQuery('#sendbutton').click (function () {
16   var text = jQuery('#chatInput').val();
17   jQuery.post('/path/to/postmessage', text);
18 });
19
20 // initial call to ajaxCall() when page is loaded
21 $(document).ready(ajaxCall);
```

Listing 2.1: A simple Ajax polling chat client using jQuery

2.1.3 Ajax Drawbacks

The technologies combined under the term Ajax add a lot of new possibilities for application developers. Unfortunately, endless polling is rather computing expensive and occupies resources on both – the server’s and the client’s side. Additionally, polling generates a lot of network traffic. This load can be shortened by decreasing the poll

frequency, but a decreased poll frequency would also lead to increased response times of the whole application. To provide a responsive application for a lot of users one needs to fine tune polling frequencies very carefully. A too low polling frequency will upset users because of long response times and a too high polling frequency will waste resources. Even with low polling frequencies, Ajax adds a high communication overhead which could be avoided.

2.2 Comet

2.2.1 What Is Comet?

Polling Ajax eliminated the need to reload the whole web page every time someone needs to see new content, but it introduced also a whole new set of scalability problems for the infrastructure of an application.

To eliminate some of those drawbacks a set of new techniques evolved to replace Ajax in scenarios where server-push was needed. These technologies are combined under the name Comet, but is also known as reverse-Ajax, Ajax-push and HTTP-streaming. They use the capabilities of web browsers to establish a persistent connection from the browser to the server and enable the server to push data to the client without a need for endless polling.

There are several ways to implement this behavior. The most common one is called long polling. There are also other possibilities like hidden iframes or multipart XMLHttpRequest, but they tend to have negative side-effects on some browsers and are therefore not recommended [Holdener, 2008].

Using long polling, the client sends an asynchronous HTTP request to the server. Once the connection is established, the server keeps it open. It is only closed, when the server needs to push data to the client. In this case, the server responds to the HTTP request with the new data. On the client, a callback is executed when the request returns. In this callback, the client instantly sends a new request. This way, there is always an established HTTP connection between the client and the server. There is no need to make any calls to the server when there is no new data available. If there is no data, the connection will be left open.

2.2.2 Example Using Comet

Listing 2.2 shows a simplified example of a long polling Comet client. The only difference to the Ajax example is that the client instantly opens a new connection after a call returns with a response. On the server side, the application needs to be implemented in a way that it only responds to a request when there is new data available.

```
1 function cometCall() {
2   // get new messages from server, register callback
3   jQuery.get('/path/to/newmessages', function callback(response) {
4     // instantly make new call
5     cometCall();
6     addNewMessagesToChat(response);
7   }, 'json');
8 };
```

Listing 2.2: A simple Comet chat client using jQuery

2.2.3 Comet Drawbacks

Comet reduces the communication overhead and response times when compared to Ajax. On the other side, it still requires the client to make a new HTTP request every time data is transferred. In cases where latency is a really big issue, like games, this could become a problem. An other drawback of Comet is the missing communication link from the client to the server: Every time the client needs to send data to the server, it needs to make an Ajax call.

Even if Comet helps a lot by reducing latency, it is still considered a “hack” by some people [Ubl, 2010].

Technologies that enable the server to send data to the client in the very moment when it knows that new data is available have been around for quite some time. They go by names such as “Push” or “Comet”. One of the most common hacks to create the illusion of a server initiated connections is called long polling.

Like Ajax, Comet is affected by the same-origin-policy [MDN, 2010] of modern web browsers. This policy only allows XMLHttpRequests to the server which served the JavaScript in the first place. Cross-domain requests to other servers are not executed by the browser and therefore not possible.

2.3 WebSockets

2.3.1 What Are WebSockets?

For years, web developers needed to use tricks like Ajax or Comet to get fast response times and desktop-like features in their web applications. But, together with the upcoming HTML5 standards, there was also new protocol introduced to provide a standardized and efficient way to address these needs. This protocol is called WebSockets [Hickson, 2010].

The specification describes a protocol which allows a full duplex socket connection between client and server that is not limited by the same-origin-policy. This connection is tunneled through HTTP(S) and is kept open until either the client or the server closes it. This enables both, the client and the server, to send data to the other entity at any given time with a fairly low response time and less communication overhead than other solutions.

To provide a consistent way of dealing with WebSockets, the document also specifies an asynchronous JavaScript API for the client side.

2.3.2 Example Using WebSockets

In Listing 2.3 a simple chat client using WebSockets is shown. The API itself is completely event driven. This means, there is no polling and there are no blocking API calls. Instead a developer registers event handlers which are called by the runtime when special events occur.

The demo code registers three event handlers: *onopen*, *onerror* and *onmessage*. The *onopen* handler, is called when the WebSocket connection is established after a connect. *Onerror* is called when an error occurs and *onmessage* is called every time a new message from the server is pushed to the client. There is a last handler which is not used in the example called *onclose*. It is called when the connection link is closed by the server.

The example also uses the *send()* method which allows to send arbitrary String data to the server. This data could be serialized JSON, XML, plain text or even Base64 encoded binary data, like pictures.

The click handler for the send button used in the previous examples is re-used and utilizes the WebSockets API instead of Ajax calls.

```
1 var connection = new WebSocket('ws://websocket.host.tld/chat');
2
3 // when the connection is open, send some data to the server
4 connection.onopen = function () {
5     connection.send('connect');
6 };
7
8 // log errors
9 connection.onerror = function (error) {
10    console.log('WebSocket Error ' + error);
11 };
12
13 // process messages
14 connection.onmessage = function (message) {
15    addNewMessagesToChat(message.data);
16 };
17
18 // register click handler to send new message
19 jQuery('#sendbutton').click (function () {
20    var text = jQuery('#chatInput').val();
21    connection.send("message: " + text);
22 });
```

Listing 2.3: A simple chat client using WebSockets

2.3.3 Drawbacks

WebSockets are still a really new technology and the upcoming standard is still under development. So there are still a lot of browsers out there which do not support them.

An other issue is that most proxy servers currently are not aware of WebSockets. Because of the possibility of cache poisoning attacks due to HTTP proxy servers which are not able to interpret the WebSockets handshake correctly, the Mozilla Firefox team even decided to disable WebSockets [Heilmann, 2010] in their new upcoming release of Firefox 4.

2.4 Put Them All Together

WebSockets are definitely the most promising technology in terms of realtime web applications. But they are also a really new technology and only supported by the latest generation of browsers. Browsers like Internet Explorer by Microsoft have no support for WebSockets at the time of writing.

To be able to write realtime applications for all kinds of browsers, there is a need for a consistent, WebSocket-like API which handles a fallback to alternative technologies like Comet, Flash-Sockets or even polling Ajax.

Fortunately there is a library called Socket.io [LearnBoost, 2010] which consists of a client side JavaScript implementation and multiple server side implementations for different runtimes and frameworks. It combines all described technologies and provides an abstraction which always chooses the best technology that is supported by the client.

3 Problems With Realtime Web Applications

The client side improvements and technologies described in Chapter 2 – especially Comet and WebSockets – add a lot of new possibilities to web based applications.

This chapter will explain the changes introduced by these technologies and the implications they have on a server side application.

3.1 Blocking Calls

In most web applications, there are not many computations which stall the system for a long time. Web application systems are mostly stalled by I/O calls. The article “What Your Computer Does While You Wait” [Duarte, 2008] shows some interesting figures in terms of time needed to access different computer subsystems:

- L1 Cache: 1 nanosecond
- L2 Cache: 4.7 nanoseconds
- Main Memory: 83 nanoseconds
- Network Access: 2 milliseconds
- Disk Seek: 15 milliseconds
- Internet Access: 80 milliseconds

While access to the CPU cache and the main memory of the computer is fairly fast, disk and especially network access need much more time to complete than memory or CPU cache access. These calls are therefore called blocking calls.

Unfortunately, most web applications need to read HTML templates, query a database and render the template to answer a request. These are also mostly blocking operations which are stalling the current thread of execution until they are completed. During

this waiting period, the current thread is blocked even if something more useful could possibly be done during this time.

When dealing with realtime web applications every user connected to an application which is using blocking I/O means blocking the whole thread of execution while the user is connected. This also means that an application using blocking I/O needs more than one thread or process to serve multiple users.

3.2 The End Of A Simple Request/Response Model

For years, web applications were basically doing the same thing: They responded to a request with the same repeating pattern.

1. Receive the request
2. Process the request
3. Render some content
4. Respond to the request with the rendered content

This very simple model was used for a lot of cases and is also the basis for Ajax applications with polling JavaScript. Unfortunately this model does not work well in realtime applications which need to be able to communicate with the server in an asynchronous way which does not tolerate high latencies.

The classical request/response model only allows clients to request data from the server but does not support pushing data from the server to clients. But for realtime applications like games, collaboration software, or chat applications, the ability to push data to clients is essential. Therefore long polling solutions, which provide at least a logical server-push, or WebSockets are needed to fulfill this requirement.

3.3 Statefulness

HTTP as the standard protocol of the Web has been designed to be fundamentally stateless. The RFC specification of HTTP 1.1 [Fielding et al., 1999] describes it this way:

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers.

WebSockets and Comet connections, on the other side, are inherently stateful and cannot be treated like usual HTTP services because of this major difference. The state of current connections has to be stored on the server, the connections are long living and the server needs the ability to broadcast data to all clients at the same time. These requirements lead a system which does not scale out without any problems.

3.4 Long Living TCP Connections

With the end of a simple request/response model also comes the end of short lived TCP connections which are only established while a request is processed and immediately closed after the response to a request has been sent to the client. In modern realtime applications there are usually many open TCP connections a server has to handle. Depending on the application there is, at least, one connection per client waiting for data to be pushed from the server.

With all these long living connections there is not only a need for new architectural decisions in the application itself but also for application servers and middleware which need to be able to handle a lot of established TCP connections.

3.5 Bidirectional, Asynchronous Communication

JavaScript on the client side was designed to be a simple scripting environment for GUI applications. It is also inherently single threaded. To avoid freezes of the UI and to provide a simple system for managing user interaction, JavaScript has an asynchronous and event driven programming model.

Not only the client side programming model, but also the interaction of users with GUI systems is asynchronous. Additionally users not only want to be notified from a server

when some new data is available, they also want to participate and to be able to send data themselves.

For an application which should be able to interconnect a lot of users it is therefore essential to support both, asynchronous and bidirectional connections.

4 Server Side

Realtime web applications need to be able to handle different traffic patterns than classical web applications following a request/response model.

This chapter will examine models to handle this traffic on the server side, look at their strengths and weaknesses and explain them using code examples.

4.1 Processes

4.1.1 Process Basics

A process in computing is a running instance of a computer program. It contains the execution code of the program and its own resources, like the execution stack and the heap, where dynamic memory is allocated.

Multitasking capabilities of modern operating systems allow to schedule several of such processes: Because a CPU is only able to execute one single task at a time, the operating system needs to schedule CPU time to allow several tasks and therefore multiple processes to run.

If there is more than one logical CPU, the operating system is able to schedule these processes so they run in parallel – each on one logical CPU. Otherwise and when there are more running tasks than CPUs available – which is the normal case – some processes are put on hold while others run. This feature, supported by every modern general purpose operating system, is called preemptive multitasking.

A running process could be anything from a daemon providing access to the file system over a network connection to a word processor. In the case of this thesis only processes providing services over HTTP, like web- or application-servers, will be considered.

Scheduling these processes comes not without any cost: With every switch between some tasks, the CPU registers, the stack pointer, the program counter and information about open files of I/O devices in use have to be saved and replaced with the ones of the next scheduled task. This action is, with each switch, restoring the context of the scheduled task and is therefore called context switch. Context switching and the CPU cycles needed by the scheduler lead to a considerable scheduling overhead when a lot of processes are running on a machine.

4.1.2 Handle incoming connections with one process

To provide HTTP services over a network, the process needs to be able to accept incoming network connections, do its work and then respond to them. In the case of the example chat application, the client would at first request static resources, like HTML, pictures or CSS files and the JavaScript file containing the client side application.

To respond to a request, the server needs to accept incoming client connections. After that, it needs to parse the request and get the requested file name. When this is done, it has to read the requested file into memory and, finally, respond to the request with the content of the file. A simplified implementation in JavaScript with a pseudo API is shown in Listing 4.1.

```
1 var listeningSocket = http.socket.listen(8080);
2
3 // loop endlessly
4 while (true) {
5   var request = listeningSocket.accept();
6   // process is blocked until a client connects
7   var requestedFile = request.getRequestedFileName();
8   var data = io.file.open(requestedFile).read();
9   request.respond(data);
10  request.closeConnection();
11 }
12 // the application will never
13 // leave the loop
```

Listing 4.1: A simple webserver running in a single process

After the resources are loaded, the client side JavaScript will try to connect to a WebSocket and will idle until either the user enters some text or there are messages pushed from the server. As seen in Listing 4.2, the server process is not only limited to serve one client at a time, it is also completely blocked when a WebSocket client connects. In

this state, the server will not be able to respond to any requests and the only way to change this would be for the client to disconnect.

In such a setup, the desired chat application would not be able to work properly because the server is, under no circumstances, able to handle more than one user at a time.

```
1 var listeningSocket = http.socket.listen(8080);
2
3 while (true) {
4   var request = listeningSocket.accept();
5
6   if (request.webSocketConnection()) {
7     while (!request.connectionClosed()) {
8       var data = processData(request);
9       // respond but keep connection open
10      request.respond(data);
11    }
12    log('The client closed the connection');
13  } else {
14    var requestedFile = request.getRequestedFileName();
15    var data = io.file.open(requestedFile).read();
16    request.respond(data);
17    request.closeConnection();
18  }
19 }
```

Listing 4.2: A webserver able to handle “normal” and WebSocket connections

4.1.3 Handle incoming connections with many processes

Since a chat service needs to be able to handle, at least, a few dozens of users, the approach with only one process doing blocking I/O is not desirable. What works very poorly when it comes to a classical request/response model, does not work at all when long living TCP connections are involved.

One thought that comes to mind when thinking about a simple model to scale up to a few dozen users would be “If one process is only able to handle one user, why not create a central dispatcher and a few dozen worker processes to handle the incoming load properly?”

Technically this model is used very commonly for normal web servers. A web server implementing this approach would be the unicorn application server for Ruby applications.

An explanation of the most important features can be found in Ryan Tomayko's Article "I like Unicorn because it's Unix" [Tomayko, 2009].

Forking independent worker processes has two major benefits: The operating system is able to schedule them without problems and there is no need to worry about unwanted side effects because there is a clear memory separation for each process provided by the OS kernel.

When considering that current servers and even desktop machines or laptops are shipped with multiple CPU-cores, the scheduling part gets even more interesting. The operating system is also able to schedule these processes using multiple cores and hence able to utilize the computing power of all available resources.

```
1 var listeningSocket = http.socket.listen(8080);
2 var request;
3
4 // are we in the dispatcher or a worker process?
5 if (process.isChildProcess()) {
6   if (request.websocketConnection()) {
7     serveWebSocketAndDie();
8   } else {
9     serveHTTPAndDie();
10  }
11 } else {
12   while (true) {
13     // block until client connects
14     request = listeningSocket.accept();
15     // fork new worker process
16     process.fork();
17   }
18 }
```

Listing 4.3: A dispatcher to handle multiple HTTP and WebSocket connections

```
1 // process websocket calls
2 function serveWebSocketAndDie() {
3   while (!request.isFinished()) {
4     var data = processData(request);
5     request.respond(data);
6   }
7   // connection closed, worker kills itself
8   process.exit(0);
9 }
```

Listing 4.4: A function which kills the worker process after the connection is closed

4.1.4 Process Optimizations

Pre-forking Forking off a new process is an expensive task on most operating systems. Considering that, most servers implementing a “one process per user”-model fork off a configurable amount of worker processes when they are started to avoid the cost of forking a new worker every time a new request needs to be handled. A new process is only forked, if all other workers are currently busy.

This approach reduces the cost of creating and initializing worker processes and leads to better response times since most clients can be served by already running worker processes.

Pooling Additionally, there is also the possibility to generate a worker pool to re-use currently running processes instead of letting them die after the client has been served. In a classical request/response model, the task of scheduling could be managed by the operating system kernel: It is a common pattern to share a listening socket between some worker processes and let the OS kernel handle the load balancing. In the scenario of a combined HTTP and Comet or WebSocket server, the dispatcher also has to differentiate if a request is a short living HTTP request or a long living Comet or WebSocket request. So the load balancing has to be managed by the server software instead of the OS kernel.

Both optimizations – pre-forking and pooling – reduce the need to fork off new worker processes, but also introduce a new level of complexity because the application not only has to fork off new workers but also has to keep track of the workers’ state to be able to schedule them properly. In addition, forked worker processes have to be observed. The ones which are not needed any more have to be shut down to free resources.

For a pure Comet or WebSocket server, the cost of managing a worker pool would possibly not pay off. For a combined HTTP, Comet and WebSocket server a worker pool for short living connections and a fork/die mechanism for long living connections would possibly be a working solution. In this setting, short living connections could be served by re-usable pooled processes to avoid creation and initialization overhead. Long living connections in such a model could be served by newly created processes to simplify the process management.

4.1.5 Constraints Of Processes

Memory Overhead Spawning worker processes itself is not as expensive as it used to be. On modern platforms like Linux, the *fork()* system call is implemented using copy-on-write pages [Manpage, 2008b]. This leads to less memory consumption since memory pages can be shared between processes. On the other side, there are still two major drawbacks: Linear increasing memory and scheduling overhead. Even if adding a new worker per user does not mean to copy a whole process, it still adds some overhead with every new user.

Memory overhead is an issue for normal web servers with a lot of users, but becomes a big problem when there are Comet or WebSocket connections involved: Every client of the web application needs to be served by a dedicated process as long as the connection between client and server is established. This means that even for inactive clients a process needs to run and occupies resources.

Scheduling Overhead Besides the memory overhead, another big problem is the amount of CPU time needed to schedule a lot of processes. Every new joining user means to add a new worker process the kernel of the operating system has to schedule. This works well when the number of processes the kernel has to schedule is fairly low. It, however, becomes a big problem when there are so much processes that most CPU cycles are needed to schedule them and almost none are left to do the actual work. At this point, the application freezes and is not able to respond to all users with a reasonable delay.

Sharing State In the example chat system, the chat server needs to be able to broadcast incoming messages to all connected clients. An incoming message needs to be forwarded to all other worker processes to provide such a functionality. To do that, the worker which received the message needs to perform inter process communication (IPC) pass on the message. This is possible to implement using system calls like *pipe()* but it adds a lot of complexity to the application. In addition, IPC also leads to more context switches because it needs to be handled in kernel mode.

Scale Out As described before, handling every connection with a single process has the benefit of being able to utilize all available hardware resources on a single machine because the kernel of the operating system is able to schedule processes so they run in

parallel on different CPUs. Unfortunately, this model is not able to automatically scale out to a network of machines when a single machine is not sufficient any more.

It is possible to scale out to more than one machine by using load balancing techniques, but the inter process communication has to be done over the network to connect processes on different machines. This also adds a considerable amount of complexity to the application.

4.1.6 Conclusion

Handling connections with one process per client connection was one of the first ideas for writing server software. It has some striking benefits: Every process is able to use predicible and easy understandable code. Modern operating systems are separating processes on a memory level so there is also no need to worry about memory corruption. These processes are scheduled by the operating system so they can be executed on multiple CPUs without any problems.

On the other hand, using a process for each connection has its limits: Using a lot of processes drastically increases memory and scheduling overhead. This leads to situations where servers are using lots of resources only to serve WebSocket or Comet clients which are idle most of the time.

Processes serving WebSocket connections also need to use system calls to communicate with each other because direct communication via shared memory is prevented by memory separation of the operating system. This leads to even more overhead because every IPC call into the kernel leads to a context switch. This situation gets even worse when the application needs to scale out to a network of servers.

4.2 Threads

4.2.1 Threading Basics

Running concurrent tasks using multiple processes can lead to memory and scheduling overhead when a lot of parallel tasks have to be executed. To reduce this overhead, modern operating systems introduced new primitives called threads. A thread of execution is the smallest unit of processing an operating system is able to schedule. The most common use case is a single threaded process: A process with one running execution thread. In this example, the process is used to group resources together and the thread is the actual entity scheduled for execution on the CPU.

Threads are mostly scheduled preemptive. Which means that a scheduler decides which thread to run at a given time. The scheduler is also responsible for suspending and resuming the execution of threads and has to decide how long each thread will be executed before another thread will be scheduled.

In contrast to a process, a thread is not able to run independently. It has to run as a subset of a process. Threads also have very little “thread local” memory and an own execution stack but share all other resources with the process that spawned them.

In addition, a thread normally has some information like a thread ID, a priority and an own signal mask. In general, context switching between threads needs to save and restore less context information compared to a context switch between processes. While switching between threads only the program counter, the stack pointer and the general purpose registers of the CPU have to be saved and restored. This leads to faster context switching compared to processes.

The faster context switching and lower memory consumption make threads much more lightweight than processes. Additionally, threads running in the same process share the same resources and are therefore able to communicate with each other via shared memory. This also leads to less overhead because IPC via kernel APIs is not needed.

4.2.2 Native Threads vs. Green Threads

When talking about threads, one needs to make a distinction between native and green threads: While native threads are mostly implemented on top of kernel threads and are

scheduled by the kernel of the operating system. Green threads, on the other hand, are scheduled by a virtual machine or a threading library and run in user space.

Both threading implementations have their own up- and downsides. Native threads are generally more expensive in terms of context switching, but the kernel is able to utilize multiple CPUs so they can run in parallel. Green threads run mostly on only one kernel thread and are therefore limited to only one CPU. On the other side, green threads are usually more lightweight than native threads and generate less scheduling overhead.

4.2.3 N:M Threading vs. 1:N Threading vs. 1:1 Threading

Most threading implementations use a 1:1 threading model. Which means that every user thread is mapped to a kernel thread. As discussed in Section 4.2.2, multiple green threads are normally mapped to one kernel thread. This mapping is also known as 1:N threading. There are also implementations, called N:M threading, where green threads are mapped to multiple kernel threads to utilize all logical CPUs.

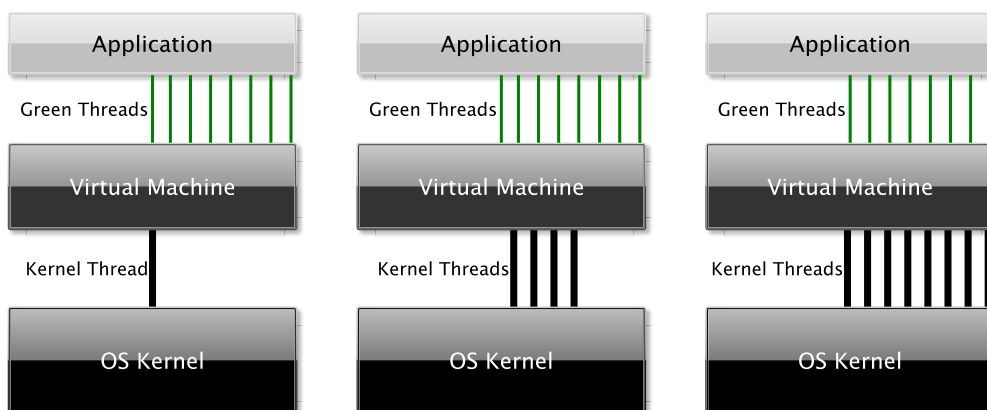


Figure 4.1: From left to right: 1:N Threading, N:M Threading, 1:1 Threading

To minimize context switching and to embrace parallelism, the number of running kernel threads should be the same as the number of available CPUs. Using green threads with a N:M threading model seems to be a good way to realize this. This also adds a lot of complexity to the threading implementation.

4.2.4 Example Using Threads

Listing 4.5 shows an example implementation of a multi threaded HTTP and WebSocket server. This example is, like the other examples, using a blocking call to *accept()*. When a new client connects to the server, a new worker thread is spawned. After that, the request is completely processed in its own thread and the dispatcher is free to process new requests.

In a multithreaded application it is possible to use the optimization possibilities described in Section 4.1.4: Pre-forking and pooling. Since threads are usually cheaper than processes, these approaches do not add a lot of improvements to the application but add more complexity.

```
1 var listeningSocket = http.socket.listen(8080);
2
3 while (true) {
4   var request = listeningSocket.accept();
5   // create a new thread
6   var workerThread = new Thread(function runnable() {
7     // request dispatching
8     if (request.websocketConnection()) {
9       serveWebSocketAndCloseThread();
10    } else {
11      serveHTTPAndCloseThread();
12    }
13  });
14  // run thread
15  workerThread.run();
16 }
```

Listing 4.5: A dispatcher to handle multiple users using threads

4.2.5 Constraints Of Threads

Race Conditions Running parallel tasks on top of threads and a shared memory model poses a serious threat to the consistency of this shared memory.

The order in which multiple threads access the data is non-deterministic. There is no guarantee that a value read by one thread is equivalent to the value an other thread reads a few clock cycles later. There is also no guarantee that data written by one thread is not overwritten by another thread at the same time. This unpredictable behavior leads to software in which it is not decidable whether a result is actually correct or if it was

corrupted due to unprotected access to shared memory. A more extensive example explaining this issues can be found in “Finding parallelism - how to survive in a multi core world” [Jäger, 2008].

Synchronization Costs As described in Section 4.2.5, the access to shared data has to be protected in order to maintain its consistency. This process of serializing the access to shared data is called synchronization. Most programming languages and threading libraries provide synchronization mechanisms using locks. Such a locking mechanism normally uses a “mutex”¹ object and the two functions lock and unlock to ensure critical sections of parallel executed code are executed atomically.

When a critical section is entered, the running thread acquires a lock on the mutex so other threads are forbidden to enter sections protected by the same mutex. This locking also has its downside: Other threads are usually waiting in a spinlock repeatedly checking if the lock is available again. If the mutex is locked for a long duration, these spinlocks become rather wasteful in terms of CPU time.

Deadlocks A system working with threads and locks is prone to deadlocks. A deadlock is usually a state where two or more threads are waiting for a lock to be released that they themselves wish to acquire. This circular locking chain leads to a situation in which a program is not able to terminate itself but has to be terminated by the outside or, otherwise, keep forever in a locked state unable to perform any work.

Deadlocks are especially problematic, because there is no general solution to avoid them.

Linear Memory Consumption When a thread is created, it also needs its own stack to store local variables and to be able to keep track of function calls and return statements. To avoid the overhead of dynamically resizing this memory segment, the size of the stack is usually predefined. All newly created threads allocate the same amount of stack memory.

The amount of memory a thread usually consumes, depends mostly on the threading implementation. According to the Linux Manpage of `pthread_create` [Manpage, 2008e], the default stack size for a new thread on Linux/x86-32 is 2 megabytes. On the JVM, the

¹Mutual Exclusion

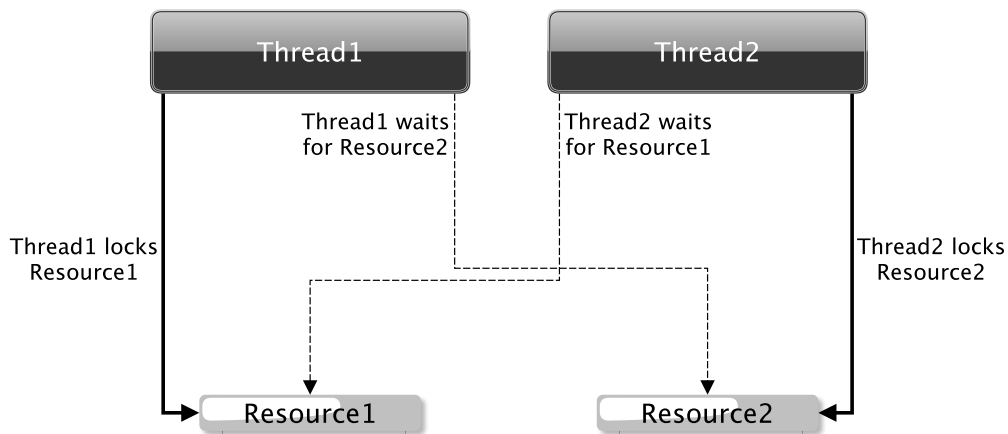


Figure 4.2: A deadlock caused by two threads

default stack size is 512 kilobytes on Sparc and 256 kilobytes on an Intel architecture. According to the Oracle Java Threading Documentation [Oracle, 2010] this size can be reduced to a minimum size of 64 kilobytes via tuning parameters of the virtual machine.

On a system with potentially thousands of concurrent users, the amount of memory consumed by threads could be, secondary to the scheduling overhead, also a limiting factor of the scalability of a realtime web application. A machine with 4 gigabytes of free memory would only be able to spawn a maximum amount of 2000 pthreads with a default stack size of two megabytes.

Scheduling Overhead The context switching between multiple threads is generally faster than the context switching between processes. In the case of the example chat application, every new user means a new running thread. The scheduling cost is increasing linearly with every new user. A server application would be, most likely, able to handle more users when implemented with threads than a version implemented with worker processes. Even if most users are not active, the resources to schedule all these threads will exceed the resources needed by the threads to do their work at some point. Threads have some performance benefit compared with worker processes, but they are also not the optimal solution in terms of scheduling overhead.

Green Threads: Global Interpreter Lock In many interpreted languages, like Python or Ruby, there is a mutual exclusion lock held by the programming language interpreter. This, so called, global interpreter lock (GIL) regulates access to the interpreter so only one thread at a time can be executed. This very coarse-grained locking mechanism protects the runtime from corrupting data in code which is not thread-safe by avoiding concurrent code execution.

With only one running thread at a time, there is no need for fine grained locking of resources. On the other side, runtimes with a GIL are not able to take advantages of multiple cpu cores and are therefore limited in their ability to scale vertically on multi-core machines. The only way to bypass this limitation is to start a complete runtime for each CPU core and loose the ability to share data between these runtimes. A more extensive explanation can be found in the article “Concurrency is a Myth in Ruby” by Ilya Grigorik [Grigorik, 2008].

4.2.6 Conclusion

Compared to operating system processes, threads are a lightweight alternative which promises to serve more users with less overhead. Unfortunately this improvement in terms of communication overhead leads to a whole set of new issues: Unprotected memory access will cause unpredictable behavior of software caused by race conditions. Protecting these memory segments with locks may also lead to the dreaded situation of a deadlock.

In the case of normal HTTP servers which respond to stateless HTTP requests, locking of mutable shared memory is not a big issue since there is usually no shared state which needs to be protected. When serving stateful WebSocket or Comet connections which need to exchange data, the situation really fast gets complicated. Edward A. Lee describes the situation in his paper “The Problem with Threads”[Lee, 2006]:

To offer a third analogy, a folk definition of insanity is to do the same thing over and over again and to expect the results to be different. By this definition, we in fact require that programmers of multithreaded systems be insane. Were they sane, they could not understand their programs.

Additionally, threads come with a scheduling and memory overhead. When compared to processes, threads come with a lower overhead but it is still to high to support a lot

of clients on a high-profile server. In general, threaded architectures with a dedicated worker thread per user are limited by the same factors which limit a system with worker processes: Linear memory consumption and scheduling overhead due to heavyweight worker entities and a lack to scale out to a network of servers.

4.3 Event Loop

4.3.1 Event Loop Basics

An event loop, message dispatcher or message loop is a programming model which allows to collect events and dispatches them inside a central loop. These events are usually collected by polling an internal or external event provider. The polling mechanism normally works in a way which blocks the whole loop until one or more messages are ready to dispatch.

There are two design patterns which describe I/O multiplexing to power an event loop: The reactor pattern described in 4.3.2 and the proactor pattern described in 4.3.3. Both patterns describe an event demultiplexer or dispatcher. Such a dispatcher is used to register handlers, watch for events to happen and execute the appropriate handler when an event has been triggered.

Event loops have been used in GUI programming for a long time. Instead of polling each available UI element for changes or even starting new threads and having to synchronize them, most UI frameworks register handlers which are executed by the event loop when the matching event has been dispatched. If one, for example, wanted to show an image when a button has been clicked, the normal approach would be to create a button and attach an event handler, or callback, to it. Under the hood, the framework registers this event handler. When the button has been clicked, this action will be dispatched by the event loop and the event handler will be executed and the image will be shown. An example of a similar framework is shown in Listing 4.6.

```
1 var button = new Button();
2
3 // register event handler
4 button.addCallback(function (event) {
5     // show desired image
6     showImage();
7 });
```

Listing 4.6: Registering an event handler in a GUI framework

Such an approach can also be used to handle a lot of TCP/IP connections to support WebSockets or Comet. Instead of actions triggered by user interaction, the actions are triggered by networking events: By using an event loop to dispatch network events and

a mechanism to register and execute event handlers, requests over the network can be handled inside one single thread.

```
1 var listeningSocket = http.socket.listen(8080);
2 var input = [listeningSocket];
3
4 function eventLoop() {
5   // event loop
6   while (true) {
7     var descriptors = io.select(input, [], []);
8     // get all readable file descriptors
9     var ready = descriptors.inputready;
10
11    // loop over all ready sockets
12    for (var i; i < ready.length; i++) {
13
14      if (ready[i] === listeningSocket) {
15        var client = listeningSocket.accept();
16        // client socket will be observed, too
17        input.push(client);
18      } else {
19        var clientSocket = input[i];
20        processRequest(clientSocket);
21      }
22    }
23  }
24 }
25
26 // start the event loop
27 eventLoop();
```

Listing 4.7: A simplified reactor event loop using *select()*

Listing 4.7 shows an example of an event loop listening for network traffic using the POSIX *select()* [Manpage, 2008d] system call. This system call blocks the entire loop until one of the observed file descriptors (FDs) identifying a network connection becomes readable, writable, a timeout was triggered or an error occurred. The interesting FDs are the ones which are readable or, in other words, the network connections where events occurred. The example also differentiates between new network connections and established connections to clients. The FDs of new connections are also put into the input array to be able to observe events on these connections, too.

4.3.2 Reactor Pattern

In short, the reactor pattern [Coplien et al., 1995] manages multiple concurrent connections by using non-blocking I/O facilities and demultiplexing events synchronously. In contrast to classical synchronous blocking I/O, calls to non-blocking I/O return immediately. An example would be writing into a network socket: An operation to that socket would return an error indicating, that the operation would block. The read/write operation to this socket itself is synchronous.

Most operating systems provide facilities to monitor such non-blocking operations. The most important ones are described in section 4.3.4. The reactor pattern formalizes this approach by specifying a way to declare interest in I/O events which indicate if a resource is ready to perform an operation. To be able to perform a read operation in a reactor system, the following steps need to be performed [Alexander Libman, 2005]:

- Register an event handler for read readiness of the desired resource
- The dispatcher waits for events
- The resource gets ready and the event dispatcher calls the event handler
- The event handler performs the actual read operation, registers other event handlers and returns control to the dispatcher

One big advantage of the reactor pattern is that it is portable since the needed facilities are available on most operating systems. There are also frameworks supporting multiple APIs. A developer using such a framework could choose the one which fits best for the usage scenario. Unfortunately the used facilities only work for network access and the reactor is also prone to blocking the event loop.

4.3.3 Proactor Pattern

The proactor pattern [Pyarali et al., 1997] is considered to be the asynchronous counterpart of the synchronous reactor pattern described in Section 4.3.2. Instead of registering handlers which are called when a resource is ready to perform an operation, an asynchronous operation is called. The call to this operation returns immediately. The operation is performed by the operating system and a completion handler is called when

the operation has finished. In the proactor pattern, a read operation needs to perform the following steps [Alexander Libman, 2005]:

- Register a completion handler and pass a user defined buffer
- Perform an asynchronous I/O operation
- While the dispatcher waits, the read operation is performed by the OS
- When the operation finishes, read data is put into the passed buffer
- The dispatcher is informed that the operation has been completed
- The completion handler is called by the dispatcher

The reactor pattern is able to perform both, network and file access, without blocking the main thread of the application. On the other hand, it needs an operating system which provides special asynchronous facilities. At the time of writing the only operating systems supporting true asynchronous facilities are Sun Solaris and Windows NT and its successors.

4.3.4 Underlying UNIX APIs

Dan Kegel describes in his article “The C100K Problem” [Kegel, 2006] the most important approaches to power a server which is able to support over ten thousand users at the same time. He also describes the most important UNIX APIs used to handle non-blocking network connections.

In general these APIs provided by the operating system kernel all provide a similar functionality: I/O multiplexing by handling non blocking I/O facilities: They observe a set of file descriptors for changes, buffer data and provide a mechanism to inform the caller when the state of one or more of these file descriptors change or a timeout occurred. To inform the caller often means to block the current thread until something happens and then return a set of changed (readable, writable) file descriptors.

select Select [Manpage, 2008d] is a system call implemented in UNIX-like and POSIX compliant operating systems. Select and its successors poll and epoll can be used to implement a reactor pattern.

Select is able to monitor the status of a set of file descriptors of open input or output channels. In the standard C API, select takes five arguments: The maximum file descriptor across all sets, sets of FDs checked for being ready to read, write or for error conditions and a structure containing timeout intervals. It returns the number of file descriptors changed in all three sets and -1 if an error occurred. Select overwrites the passed in sets of file descriptors. These sets need to be checked for changes when the call returns. In addition, the file descriptor sets need to be copied to re-use them because select will overwrite them. There are macros provided to manipulate the file descriptor sets: *FD_SET()*, *FD_CLR()*, *FD_ZERO()*, and *FD_ISSET()*.

poll The more modern poll [Manpage, 2008c] system call performs the same operations as select without destroying the input data. It takes three arguments: An array of structures containing watched file descriptors and bitmasks of requested and returned events, the size of this array and a timeout. It returns the number of ready descriptors or -1 when an error occurred and 0 when a timeout occurred.

epoll The only on Linux 2.6 available epoll [Manpage, 2008a] facility is a version of poll which scales well on large numbers of watched file descriptors. The epoll facility consists of three system calls: *Epoll_create*, *epoll_ctl* and *epoll_wait*. As explained in Zed Shaw's article "poll, epoll, science, and superpoll" [Shaw, 2010], epoll is not necessarily more efficient than poll. In fact, epoll is great when the total number of file descriptors is high but the total/active ratio is fairly low. When having a lot of active file descriptors, poll performs better.

AIO POSIX asynchronous I/O (AIO) [Group, 1997] is a specification which truly asynchronous event processing by providing APIs for network connection and file access which queue a signal when an operation is completed. Instead of demultiplexing events in a synchronous way, by signaling when a file descriptor is ready to read or write, AIO executes the operation and calls a completion handler when the operation has finished. It can be used to implement a proactor pattern described in Section 4.3.3.

4.3.5 Example Using An Event Loop

An example implementation of the described chat service using the WebSocket/Comet API provided by the Socket.io library for Node.js is shown in Listing 4.8.

The example shows a web server delivering an HTML file and a Socket.io server on top of the web server connecting all clients. The server instance is the entry point of the whole application. The given callback function is called every time a client connects. If this connection is a WebSocket or Comet connection, it is transparently handled by the Socket.io library.

The example also shows the usage of asynchronous APIs for reading files to avoid blocking the event loop. In this case, the *fs.readFile()* call returns immediately and executes the callback function when the file has been read. The client connection requesting this file can be suspended until the callback is triggered. In the meantime, the server is able to serve other clients until the file has been read from disk.

The Socket.io server is also structured in an event driven manner: A callback function is executed every time a new WebSocket or Comet client connects. When this event is triggered, another handler is registered. If this particular client sends a new message, the event handler is called and broadcasts the message to all the other connected clients.

```
1 var http = require('http');
2 var io = require('socket.io');
3 var fs = require('fs');
4
5 function getHTML(callback) {
6   fs.readFile('index.html', 'utf8', callback);
7 }
8
9 var server = http.createServer(function(req, res){
10   // async read html file from disk
11   getHTML(function (err, htmlContent) {
12     // handle error
13     if (err) {
14       log(err);
15       showErrorPage(res);
16     } else {
17       // write HTTP header
18       res.writeHead(200, {'Content-Type': 'text/html'});
19       // return actual content
20       res.end(htmlContent);
21     }
22   });
23 });
24
25 server.listen(80);
26
27 var socket = io.listen(server);
28
29 // incoming client
30 socket.on('connection', function (client) {
31
32   // incoming message, broadcast it to all clients
33   client.on('message', function (message) {
34     socket.broadcast(message);
35   });
36
37 });
```

Listing 4.8: A WebSocket server using node.js and the Socket.io library

4.3.6 Constraints of an Event Loop

Inversion Of Control Inversion of control is a design pattern describing a software architecture where control flow is not entirely controlled by the code written by a developer but by a framework. Instead of structuring an application by writing sequential code, a developer defines subroutines which are called by the framework.

Since an event loop is an inherently event driven system, there is no way for a developer to define a sequentially executed control flow. Coming from a background of procedural programming this may seem strange to some programmers because they need to “write their code inside out”. In the beginning of 2011 there was a heated discussion on the Node.js mailing list with the title “I love async, but I can’t code like this“ [Mailinglist, 2011] covering the problems of new users with the concept of inversion of control in an event loop system.

Hard To Debug Applications running on an event loop tend to be harder to debug than sequential ones. This has mainly two reasons: Because of inversion of control, the event system determines when code is executed. The other reason is that exceptions thrown inside event handlers are caught by the event loop. Which means that stack traces are not linked to the code which actually made a call and registered an event handler but only to the event loop and the call stack inside of the event handler. To avoid such situations, the error handling also needs to be done in an event driven way. One possibility is to register error handlers in addition to event handlers. An other possibility is to catch exceptions and pass them to the event handler. Both are not as easy to debug as sequentially executed code which allows to trace the full call stack.

Blocking The Event Loop As seen in Listing 4.7 the entire thread, and therefore the entire application, is blocked while event handlers are executed. If one of these handlers contains code which is performing blocking operations like reading a file or querying a database with synchronous calls, the whole application hangs and performance decreases dramatically. To avoid these problems, only asynchronous/non-blocking I/O operations should be used inside of an event loop.

An other problem are long taking CPU intensive calculations. To prevent blocking the event loop, these calculations should be done by another process or another thread.

Single Threaded The concept of event loop systems is inherently single threaded. This is on one side a big benefit because this concept is able to support thousands of incoming network connections within a single thread: There is no need to worry about memory corruption or locking strategies and it consumes an almost linear amount of memory no matter how many network connections are served

On the other hand this limitation makes it harder to scale realtime web applications vertically: A single threaded application can not use more than one CPU and therefore not utilize all available resources on server with multiple CPUs.

4.3.7 Conclusion

An event loop is a powerful abstraction which allows to handle a lot of connections within a single thread and with fairly low memory and scheduling overhead. It powers new high performance servers like the NginX² and Lighttpd³ HTTP servers or the Redis⁴ NoSQL database.

However, a programmer using this abstraction needs to be aware that blocking the event loop leads to heavily decreased performance and therefore blocking calls inside the event loop or callbacks should be avoided at all cost. There are several ways to achieve this goal: Frameworks like Node.js do only allow very few blocking I/O calls and make the programmer aware that it may have a negative impact on the performance of their applications if they use them. Other frameworks, like Ruby EventMachine, provide a thread pool to execute otherwise blocking operations on threads – mostly guarded by the global interpreter lock of the underlying runtime.

Programmers developing server application also need to rethink in terms of control flow and code structure because of the heavy use of inversion of control. GUI developers and web developers using client side JavaScript on the other side are already used to this kind of programming.

One of the biggest problems in terms of scalability is that an event loop is a mechanism which provides concurrency but not parallelism. In other words: The single threaded design of event loop applications leads to a situation where only a single CPU can be utilized to serve all clients concurrently. This is no real problem when handling stateless protocols like HTTP: Using a load balancing mechanism and creating one process per CPU with its own event loop makes it really simple to scale simple HTTP servers both horizontally and vertically by distributing requests over a set of running processes or machines.

²<http://nginx.org/>

³<http://www.lighttpd.net/>

⁴<http://redis.io>

WebSockets and Comet, on the other side, are stateful and data needs to be exchanged between users and therefore between the processes and machines they are connected to. For a chat application with only a few hundred users this should be not a big problem since supporting them is possible with only one CPU core. For a chat service hosting thousands of chat rooms this is certainly problematic. For such big applications the load balancing should make sure that users on the same chat rooms are served by the same process. Or more generally speaking: Users who need to exchange data should be tightly grouped together. When latency and communication overhead are no big problem then using a message queueing system to connect all running processes on different servers may also be a reasonable approach to scale out realtime services.

In general, using an event loop to power realtime web application is a good approach to power small applications but needs complex load balancing mechanisms or third party message passing solutions to scale up or to scale out.

4.4 Actor Model

4.4.1 Actor Basics

The actor model offers a completely different approach to concurrency and parallelism than processes or multithreading: It follows the principle “Do not communicate by sharing state; share state by communicating” [Grigorik, 2010] and is based on the mathematically founded model of “Communicating Sequential Processes” [Hoare, 1983].

This approach is sometimes also called “message passing concurrency” or “share nothing concurrency”, but in general all names describe the same pattern: An actor uses immutable messages to communicate with other actors instead of passing around references to mutable data structures. A benefit of such a model is the de-coupling between actors: Instead of method invocation and memory access only messages are sent between otherwise cohesive actor instances.

A good analogy explaining the actor model can be found in “Programming Erlang” [Armstrong, 2007]:

An Erlang program is made up of dozens, thousands, or even hundreds of thousands of small processes. All these processes operate independently. They communicate with each other by sending messages. Each process has a private memory. They behave like a huge room of people all chattering away to each other.

The actor model uses basically three building blocks to accomplish its goals:

- Very lightweight, isolated processes (Actors)
- Immutable data structures (Messages)
- A lock-free queueing mechanism (Queues)

Actors An actor is an entity which can receive messages and perform operations based on that message. On a system level actors can be seen as very lightweight processes which share no properties whatsoever with other actors. They are therefore completely isolated from each other. Each actor is only able to access its own state and is instrumented via messages. Mutable state is encapsulated inside of an actor and the only possibility for other actors to mutate this internal state is by sending messages to the actor.

As an effect of this isolation, actors are well suited for highly concurrent and fault tolerant systems: Decoupled, isolated actors can easily be replicated and distributed to fulfill high concurrency and fault tolerance requirements. Actors are not only decoupled from each other but they are also decoupled from operating system processes. This feature enables an actor implementation not only to schedule actors on multiple CPUs, but also to schedule them on different physical machines of a computer network.

Actors are normally lightweight enough to enable an application to create millions of them on commodity hardware. To do something useful, a programmer has to split up an application into logical units which communicate via message passing in an asynchronous fashion.

The actor itself is scheduled by its runtime or special libraries because there is no operating system primitive to schedule an actor directly.

Messages An actor system consists of lots of actors which communicate with each other. This communication is implemented via passing of immutable messages. This message passing could be implemented via reference passing – if the actors run on the same local machine. Avoiding mutable messages helps also to avoid unwanted side effects caused by concurrent access to mutable data and enables the runtime to optimize message passing by passing on references to objects instead of the objects themselves.

Queues Every operation that sends messages to an actor is asynchronous. So what happens when a lot of concurrent messages are sent while the actor is currently busy? – Most actor implementations handle this kind of situation with a message queueing mechanism: Every actor has its own queue to which messages are delivered. This queue is often called “the mailbox”. This mechanism is very similar to an event loop discussed in Section 4.3. One could argue that every actor runs its own small event loop system to manage incoming messages.

While an actor is busy working on a task, the queue is filled up. When the actor has finished its current task, the next queued message is handled. Such a message queue normally follows FIFO⁵ semantics. To allow concurrent actors to access to the message queue, it has to be locked internally to avoid memory corruption. To allow high throughput rates, the message queue could also be implemented using a lock-free data structure instead of internal locking.

4.4.2 Fault Tolerance

Actor systems were, in times of single CPU machines, not developed to enable concurrent programming. The goal of actors was to enable fault tolerant systems in the first place. The approach to create a fault tolerant system with an actor system is actually very simple: Do not avoid failure but embrace and observe it. Instead of avoiding an error scenario, important actors can be observed by a special kind of actor. When an actor encounters an error, it crashes. This crash is detected by the observer and the crashed actor will be instantly restarted. This mechanism avoids complicated error handling and also makes it possible to leave an actor always in a consistent state.

4.4.3 Remote Actors

The fundamental design of an actor system – asynchronous passing of immutable messages – makes it not only well suited for highly concurrent and fault tolerant systems. It also enables an implementation to distribute an actor system beyond the boundaries of one physical machine. Most actor implementation also provide a mechanism called remote actor which allows to scale an actor system out to a whole network of machines. The asynchronous message passing also works when an actor is listening on a socket and the messages are distributed via TCP/IP. This possibility not only makes actors interesting for distributed computing but also allows to build an architecture with redundancies to enable higher fault tolerance.

4.4.4 Actors Using “Green Processes” – Erlang Style

One of the most important programming languages to support the actor model was Erlang. Before being popular in distributed and concurrent computing Erlang was used

⁵first in, first out

to program large telephone switches and high profile applications which had to run virtually forever. The Erlang virtual machine has matured over more than two decades. It supports a very lightweight actor model which allows to schedule actors on multiple CPUs. Distributed Erlang also allows to distribute an actor system over a network of computers.

The paper “Inside the Erlang VM” [Lundin, 2008] provides some useful information and insights covering this topic.

Creating Lightweight Processes To be able to provide a lightweight primitive, Erlang uses “green processes” which are scheduled entirely by the virtual machine. These processes behave a lot like green threads with the difference that they do not share any attributes. Message passing between these processes is really fast because it happens in user space so no context switch to kernel mode is necessary. When they are created, only a small amount of memory is allocated so they are really lightweight. According to the Erlang User Documentation [AB, 2010] an actor initially allocates only 233 bytes of memory including the stack:

The default initial heap size of 233 words is quite conservative in order to support Erlang systems with hundreds of thousands or even millions of processes. The garbage collector will grow and shrink the heap as needed.

Schedule Actors On Multiple CPUs At first, the Erlang VM had no symmetric multiprocessing (SMP) support and was therefore limited to only one CPU per physical machine. In this time the virtual machine had one task scheduler and a global task queue. The scheduler picked tasks from the queue and executed one at a time.

In newer versions of the virtual machine there are multiple task schedulers which pick their tasks from a global task queue. Each of them runs on a kernel thread which is scheduled by the operating system. To protect this queue from corruption caused by race hazards, the access has to be locked by the runtime. With this mechanism the VM is able to distribute the processing load of all actors to multiple CPUs. The Erlang User Documentation [AB, 2010] describes it this way:

The SMP emulator (introduced in R11B) will take advantage of multi-core or multi-CPU computer by running several Erlang schedulers threads (typically,

the same as the number of cores). Each scheduler thread schedules Erlang processes in the same way as the Erlang scheduler in the non-SMP emulator.

Having only one central task queue becomes a significant bottleneck with an increasing numbers of CPUs. To resolve this issue, newer versions of Erlang are using one task queue per scheduler and a migration logic to distribute tasks to these queues [Lundin, 2008]. With this concept it is possible to utilize a large amount of CPUs without significant performance penalties.

4.4.5 Actors On A Threadpool

While Erlang has support for actors as a concurrency primitive in its virtual machine, other languages supporting actors have to implement them in an other way.

The Scala programming language is an example of a language with built-in actor support running on the Java Virtual Machine (JVM). This actor system is implemented on a library level and does not need special support of its runtime. As stated before, threads are heavyweight primitives which come with high creation and initialization overhead. To solve this problem, Scala uses a hybrid system which uses a dynamically resizable thread pool together with a model which unifies thread based and event based actors.

The paper “Actors that Unify Threads and Events” [Philipp Haller, 2007] describes the design choices of this actor system and shows some implementation details.

4.4.6 Example Using Actors

Actors are fast to create and to schedule and are therefore well suited to serve lots of clients. The example consists of two parts: Listing 4.9 shows an observer actor which manages all worker actors. Each time a worker is spawned, the *init* method of the worker is called and the worker sends a “workerRegister” message to the observer. The observer then stores the reference to that worker inside an object. Before an actor dies, the runtime executes its end-method. This method also sends a “workerUnregister” message to the observer so the reference is deleted from the store object. When a worker receives a chat message from its client, it sends a “chat”-message to the observer. The observer then broadcasts it to all the other workers so they can deliver it to their clients

If there is a message delivered where no case statement matches, it is logged.


```
1 // create new instance of actor
2 var observerActor = new Actor({
3   // manager which stores all workers
4   workers: new ActorStore(),
5
6   // receive function which is called on incoming messages
7   receive: function(message) {
8     switch(message.type) {
9
10    case 'workerRegister': this.workers.add(message.data);
11      break;
12
13    case 'workerUnregister': this.workers.remove(message.data);
14      break;
15
16    case 'chat':
17      // broadcast message to all active workers
18      this.workers.broadcast({type: 'chatBroadcast', data: message.data});
19      break;
20
21    default:
22      log('unknown message: ' + message);
23    }
24  }
25
26 }).spawn();
```

Listing 4.9: An actor to observe all workers

Listing 4.10 shows an example of a `WebSocketActor`. To create this kind of actor a reference to a client is needed. When a new message from this associated client is received, the runtime puts it into the actor’s mailbox. The `receive`-method is called when a new message is delivered to this mailbox.

The example `WebSocketActor` is able to receive two kinds of messages: A “`websocketMessage`” sent by the client and a “`chatBroadcast`” message sent by the observer actor. When the worker actor receives a `chatBroadcast`, the message is delivered to the client. If it receives a “`websocketMessage`”, the messages is sent to the observer.

```
1 var listeningSocket = http.socket.listen(8080);
2
3 function spawnActor(client) {
4   var workActor = new WebSocketActor({
5
6     init: function () {
7       observerActor.send({type: 'workerRegister', data: this});
8     },
9
10    // receive it called by the runtime
11    receive: function (message) {
12      switch(message.type) {
13
14        case 'chatBroadcast':
15          this.client.send(message.data);
16          break;
17
18        case 'websocketMessage':
19          // relay message to observer
20          observerActor.send({type: 'chat', data: message.data});
21          break;
22      }
23    },
24
25    end: function () {
26      observerActor.send({type: 'workerUnregister', data: this});
27    }
28  }, client);
29
30
31 return workActor.spawn();
32 }
33
34 while (true) {
35   var client = listeningSocket.accept();
36   // create a new actor and pass in a client reference
37   var actor = spawnActor(client);
38   log('spawned: ' + actor);
39 }
```

Listing 4.10: An example implementation of a chat actor

4.4.7 Constraints Of The Actor Model

The need to rethink concurrency Actor based concurrency is fundamentally different to the approach used in most “mainstream” programming languages. This may cause programmers to rethink their whole approach of software design because it is very

different to the way they were designing software for years.

In a “classical” programming model tight coupling and modification of shared memory are the normal case. Method invocations are also applied instantly and values are returned synchronously. These treasured attributes make it easy for programmers to predict the behavior of their code. And all these attributes do not apply to a highly concurrent actor system where messages are queued and only handled when they are on top of the queue.

Not only the call semantics change: It is not guaranteed that an actor is able to interpret messages sent to it correctly. It is possible that a message is received but the code to deal with this particular message is not implemented.

Complexity Constructing a program which handles complex use cases is not an easy task. This task gets even more complicated when it has to be executed in parallel. Designing a program which consists completely of actors communicating in a completely asynchronous fashion adds at least some complexity to an already complex task. For example, a read operation executed by one actor to read data from another actor would require two asynchronous messages to be sent: A message to request the data and another message to actually send the data to the requesting actor.

4.4.8 Conclusion

The actor model, or message passing concurrency in general, is a mathematically founded approach to concurrency. It goes beyond the idea of multithreading in terms of parallel executed tasks and shows a model which enables developers not only to write concurrent, but also fault-tolerant and distributed software.

Because its high availability features the actor model has been used to power high profile, fault tolerant telephone switching systems for decades. With the emerge of parallel computing it is also getting popular in general purpose computing.

The downsides of the actor model are no technical ones: The first big problem is that most “mainstream” languages do not support this model and programmers need not only to learn a new concurrency paradigm but also new programming languages. The second problem is that programmers tend to be irritated and even intimidated by this new approach which is so radical different to the techniques they used for a long time.

4.5 Coroutines

4.5.1 Coroutine Basics

Coroutines are programming components first defined by Malvin Conway in his article “Design of a Separable Transition-Diagram Compiler” [Conway, 1963]. They are a generalization of subroutines which provides multiple entry points. They also allow to resume or suspend their execution at given points. Additionally coroutines have their own call stack and can encapsulate their own current state. They can be used to implement other programming components like generators, iterators or UNIX pipes in a very efficient manner.

Compared to coroutines, subroutines can be explained as simpler versions of coroutines which only have one entry and one exit point.

As described in Section 4.2, preemptive scheduling means that a scheduler decides when a thread will be run. Contrary to preemptive scheduling, coroutines are scheduled cooperative. This means that the code running in the coroutines themselves decides when to exit or return control to other coroutines. It also means, that in a concurrent system powered by coroutines, only one coroutine is able to run at a time. Only when it suspends its current execution, other coroutines will be executed. A comparison of preemptive and cooperative scheduling is shown in Figure 4.3.

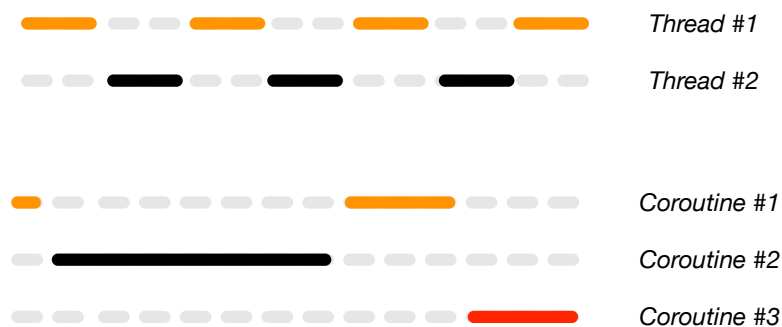


Figure 4.3: Comparison of preemptive scheduling with cooperative scheduling

Since they are scheduled cooperative and are normally implemented inside of a runtime or library, coroutines are much more lightweight than threads. Coroutines are scheduled manually, there is no need for a complex scheduler and because coroutines mostly run completely in user space, the context switching is also more lightweight. According to the

documentation of Ruby Fibers [Ruby-Documentation, 2010], a coroutine implementation for the Ruby programming language, each newly created fiber only allocates 4 kilobytes of memory:

As opposed to other stackless light weight concurrency models, each fiber comes with a small 4KB stack. This enables the fiber to be paused from deeply nested function calls within the fiber block.

The nature of cooperative scheduled entities – such as coroutines – is that they are executed serially. In this setting there is no need for expensive locking mechanisms to protect memory from corruption, because tasks do not run in parallel.

Coroutines are implemented by different programming languages and libraries in different ways. The implementation considered for this thesis is mostly known as “stackful asymmetric coroutines” as described in “Revisiting Coroutines” [de Moura and Ierusalimsky, 2004].

4.5.2 Coroutine API

There are several different implementations of coroutines. The most common one is called asymmetric coroutines which is described in “Coroutines in Lua” [Moura et al., 2004]:

Asymmetric coroutine facilities are so called because they involve two types of control transfer operations: one for (re)invoking a coroutine and one for suspending it, the latter returning control to the coroutine invoker. An asymmetric coroutine can be regarded as subordinate to its caller, the relationship between them being similar to that between a called and a calling routine. A different control discipline is implemented by symmetric coroutine facilities, which provide a single transfer operation for switching control to the indicated coroutine.

Most implementations of asymmetric coroutines provide a common set of API calls:

Coroutine.create(function) creates a new coroutine. A function with executable code is passed in as a parameter and a newly created coroutine is returned.

Coroutine.resume(coroutine) (re)invokes the execution of a coroutine. The coroutine itself is passed as a parameter. The *resume()* call return values yielded by the coroutine.

Coroutine.yield(value) is similar to a return statement in a subroutine. It yields any value passed in as a parameter. This value is returned by *Coroutine.resume()* to the caller. The coroutine is now suspended. A new invocation with *Coroutine.resume()* will execute the statements after the last yield call.

Coroutine.status(coroutine) returns the status of the coroutine passed in as a parameter. The status can be “running”, “suspended”, “normal” and “dead”.

4.5.3 Coroutines And Generators

Coroutines can be used to implement generators in a simple and efficient manner. In Listing 4.11 an example of a generator is shown. This generator returns a multiple of 5 with each call. At first 5, then 10, then 15 etc.

```
1 var generator = coroutine.create(function () {
2     var x = 0;
3     while (true) {
4         x += 5;
5         // pause the coroutine an yield result
6         coroutine.yield(x)
7     }
8 });
9
10 generator.resume(); // 5
11 generator.resume(); // 10
12 generator.resume(); // 15
13 generator.resume(); // 20
14 generator.resume(); // 25
15 generator.resume(); // 30
```

Listing 4.11: A generator using coroutines

Since coroutines can encapsulate their own state, the example generator stores the last yielded value in a local variable. With this encapsulated state, there is no need to recalculate the last yielded value. This value is incremented by 5 and then yielded inside

of an endless loop. At this point the coroutine returns control to the caller. On each *resume()* call, the loop is executed again and a new value will be yielded.

4.5.4 Coroutines And Cooperative Concurrency

Coroutines have the ability to suspend their execution, resume control to other coroutines and encapsulate their own state. Since they are scheduled cooperative, it is possible to use coroutine as a building block of a cooperative concurrent system.

4.5.5 Example Using Coroutines

In Listing 4.12 an example chat application using an event loop and coroutines is shown. This example consists of three parts: The event loop, an *addClient()* function and a *broadcast()* function.

The event loop uses *select()* [Manpage, 2008d] to observe a set of file descriptors stored in the input array. When one or more file descriptors become readable, a loop iterates through all of them and dispatches events. On new connections an *addClient()* function is called which creates a new coroutine for the newly connected client and adds it to the clients object. If a client socket becomes readable, the matching coroutine will be resumed. After the coroutine yields the message from the client, the *broadcast()* function will be executed to relay the message to all other clients.

```
1 var listeningSocket = http.socket.listen(8080);
2 var input = [listeningSocket];
3 // stores the coroutines for each client
4 var clients = {};
5
6
7 function eventLoop() {
8   while (true) {
9     var descriptors = io.select(input, [], []);
10    var ready = descriptors.inputready;
11
12    for (var i; i < ready.length; i++) {
13      if (ready[i] == listeningSocket) {
14        addClient();
15      } else {
16        var clientSocket = input[i];
17        var client = clients[clientSocket];
18        var message = coroutine.resume(client);
19        broadcast(message);
20      }
21    }
22  }
23 }
24
25 // start the event loop
26 eventLoop();
```

Listing 4.12: An event loop using coroutines

The *addClient()* function accepts the current connection and creates a new coroutine for this particular connection. Inside of the coroutine runs an endless loop which either sends messages to the client or reads messages from it. In both cases, execution is suspended with a *coroutine.yield()* call. In case a message has been received, this message will be yielded to the caller.


```
1 function addClient() {
2   // every client coroutine has access to its socket
3   var socket = listeningSocket.accept();
4   // new socket to observe
5   input.push(socket);
6   // create a client coroutine and add it to clients
7   clients[socket] = coroutine.create(function (message) {
8     // loop to get client input
9     while (true) {
10      if (message) {
11        socket.send(message);
12        // suspend control
13        coroutine.yield();
14      } else {
15        var clientMessage = socket.getData();
16        // suspend control and yield message to caller
17        message = coroutine.yield(clientMessage);
18        // after re-invocation, yield will return the
19        // value passed to coroutine.resume
20      }
21    }
22  });
23 }
```

Listing 4.13: The addClient function used in the event loop

The *broadcast()* function is called when a new message has to be relayed to all connected clients. It iterates over the set of stored coroutines and calls *coroutine.resume()* on all of them. Via the resume call, the passed in message will be returned by the *coroutine.yield()* call inside of each client's coroutine.

```
1 function broadcast(message) {
2   // loop over all clients
3   for (var key in clients) {
4     // resume coroutine, pass in message
5     coroutine.resume(clients[key], message);
6   }
7 }
```

Listing 4.14: The broadcast function used in the event loop

4.5.6 Constraints Of Coroutines

Single Threaded Coroutines are cooperative scheduled entities. They explicitly return control to their caller or (re)invoke the execution of other coroutines and suspend their current execution. In order to be able to run coroutines in parallel, it would be necessary

to implement a scheduler which is able to determine which routines depend on each other. This scheduler would then be able to decide which routines can be executed in parallel.

Since the implementation of such a scheduler would be certainly very complex, all coroutine implementations known to the author are single threaded. If it is necessary to achieve parallel execution, an application needs to combine coroutines with multi-processing or multithreading mechanisms.

Blocking Calls The execution flow of an application using coroutines is explicitly defined by (re)invocation and suspension of the involved coroutines. Blocking calls like file I/O or network communication in such a system would block the execution of the whole system.

Diversity As a programming construct, coroutines are not as well defined as other concepts. According to “Revisiting Coroutines” [de Moura and Ierusalimsky, 2004], there are mainly three points which distinguish different coroutine implementations:

- *the control-transfer mechanism, which can provide symmetric or asymmetric coroutines;*
- *whether coroutines are provided in the language as first-class objects, which can be freely manipulated by the programmer, or as constrained constructs;*
- *whether a coroutine is a stackful construct, i.e., whether it is able to suspend its execution from within nested calls.*

In addition to the diversity of the implementations, coroutines are also sometimes called fibers and mixed up with related concepts like continuations, generators and iterators. All this leads to confusion for developers who want to understand the concept of coroutines in order to use it.

Availability The popularity of a programming concept is highly influenced by the support of programming languages developers use on their daily basis. Unfortunately, the support for coroutines in popular programming languages is not as widespread as the support for threads. For most programming languages, coroutines are only implemented on a library level and are therefore not often used and well understood by developers.

At the time of writing, there are only two mainstream programming languages which have support for coroutines in their core libraries: Lua [Lua-Documentation, 2010] and Ruby [Ruby-Documentation, 2010]. Other languages like Scheme, C or Python provide the building blocks to implement coroutines on a library level but do not support them directly.

4.5.7 Conclusion

Even if coroutines have been available for decades, they are not known to many programmers. This situation may be a result of the diversity of this concept and the lack of coroutines as a first class element of most popular programming languages. Nevertheless they are a powerful construct which enables programmers to manage cooperative scheduling in a very efficient manner.

In general, coroutines are also more lightweight than threads in terms of memory usage and scheduling overhead and therefore faster to create [Ali, 2008].

Because they are mostly single threaded and suffer from blocking calls, it is advisable to use coroutines together with non-blocking or asynchronous I/O facilities. In this setting, the manual scheduling can be used to hide complex callback mechanisms in order to create abstractions where asynchronous calls look synchronous.

Coroutines can also be used to implement abstractions like iterators, generators or pipes, but these constructs are not directly related to the topic of this thesis.

4.6 Hybrid Systems

All introduced models have their own up- and downsides. None of them is a perfect fit for general application programming, but some can be used together to exploit synergy effects. This section will show a few examples where different concepts are used together to create more robust system.

4.6.1 Event-Loop with Threads

Event loops are built upon non-blocking I/O facilities provided by the operating systems. Unfortunately there are no such facilities for file system access on most operating systems.

To prevent locking the whole process, a dedicated thread-pool for file I/O purposes can be implemented. Instead of using blocking I/O calls to read or edit files, threads can be used to execute this otherwise blocking call. When the operation is complete, this thread can inform the caller by invoking a callback.

4.6.2 Event-Loops On Multiple Processes

Most event loop systems are implemented in a single threaded fashion and therefore not able to utilize more than a single CPU. For applications built on top of these systems is possible to create one process with its own event loop per CPU. With a distribution mechanism to dispatch incoming requests to these processes all available resources can be used.

4.6.3 Event-Loop And Co-Routines

Programming event loop system can get confusing when many I/O operations are depending on each other. This can lead to situations where asynchronous calls and a lot of nested callbacks are generating code which is hard to read and to understand.

In such cases manually scheduled co-routines enable programmers to create synchronous looking code which is easy to read and to understand but is completely asynchronous

under the hood. An example implementation of this concept using Ruby and Eventmachine is explained in the article “Fibers & Cooperative Scheduling in Ruby” [Grigorik, 2009].

4.6.4 Actors On A Threadpool

As described in Section 4.6.4 there are actor implementations on virtual machines which are not able to schedule actors directly. These implementation are mostly using threads as a low level concurrency primitive to run an actor system on top of them.

4.6.5 SEDA

The staged event driven architecture (SEDA) [Welsh, 2002] is another example of a hybrid system. It uses event loops to achieve concurrency and threads to utilize all hardware resources. SEDA decouples event and thread management from the application logic and uses different processing stages which are interconnected by event queues. To ensure a high throughput, each stage can be conditioned by thresholding or filtering the event queue. This conditioning may be necessary to fine tune the architecture of an application to ensure it fits to the application’s profile. With this mechanism it is also possible to throttle particular stages to avoid overloading the following ones.

5 Frameworks

Being able to use the described patterns and techniques may help programmers writing high-end server software. For web developers it may be the wrong approach to implement their own WebSocket or Comet server. It also would be a waste of time sine a lot of the code to power such systems can be re-used in form of frameworks.

This chapter will introduce some of the most interesting frameworks in 2010/2011.

5.1 Lift

Lift¹ is a web application framework written in the Scala programming language which runs on the Java Virtual Machine. It has integrated support to for Comet. WebSocket support is planned. Since Scala has good support of actors on a library level, Lift also supports actors inside of web applications. The special CometActors are powered by a continuations system provided by the Jetty application server.

5.2 EventMachine

EventMachine² is a general purpose event loop framework for the C++, Ruby and Java programming languages. It implements the reactor pattern. For Ruby it uses native extensions be able to run the event loop without the restrictions of the global interpreter lock. There are a lot of libraries and a fork of Ruby on Rails 3.0³ using it to build event driven software.

¹<http://liftweb.net>

²<http://rubyeventmachine.com/>

³<https://github.com/igrigorik/async-rails>

5.3 Twisted

Twisted⁴ is, like EventMachine, a framework which abstracts an event loop to provide a re-usable basis for event driven programs. It is available for the python programming language and supports pluggable reactors using different APIs like select, poll, KQueue and others.

5.4 Aleph

Aleph⁵ is a framework for asynchronous communication written in the Clojure programming language. It is running on the JVM and built on top of the Java Netty⁶ library. What makes this framework especially interesting is the underlying multi threaded Netty library in combination with a programming language which explicitly supports concurrency by providing transactional references [Hickey, 2010]. This model allows to create a WebSocket server with the integrated WebSocket library which scales in a single machine on multiple CPUs without having to worry about memory corruption.

5.5 Misultin

Another framework written in a functional programming language is Misultin⁷. Misultin is written in Erlang and runs on top of the Erlang Virtual Machine (BEAM). It is a HTTP framework for building fast and lightweight web applications using Erlang actors. It also includes a WebSocket library.

5.6 Node.js

Node.js⁸ is a general purpose event driven runtime environment for server side JavaScript applications. It uses the Google V8 JavaScript VM in combination with libev⁹ which is

⁴<http://twistedmatrix.com/>

⁵<https://github.com/ztellman/aleph>

⁶<http://www.jboss.org/netty>

⁷<https://github.com/ostinelli/misultin>

⁸<http://nodejs.org>

⁹<http://software.schmorp.de/pkg/libev.html>

a fast C/C++ library to create event loop systems. Node itself is implemented in C++ and extensible using C++ and JavaScript. It also brings its own thread pool to allow non-blocking file operations on systems which do not natively support them.

One big benefit of Node is the lack of synchronous I/O calls: Almost every I/O operation is asynchronous and is therefore not blocking the event loop. Node also includes the CommonJS module system which makes it easy to write modules. At the time of writing there are several different WebSocket and Comet implementations available. One of the most used is Socket.io which was already described in Section 2.4.

6 Conclusions

The research for this thesis has provided some valuable information regarding realtime web applications and their implementations on the client and server side. This chapter will explain the most important ones and make some assumptions regarding the future of this kind of application.

6.1 There Are No Silver Bullets

The research for this thesis showed one thing: Even if WebSocket and Comet applications have relatively similar requirements, there is no approach which fulfils the need of all applications. All shown approaches have their own up and downsides and the decision of choosing one should therefore be well considered.

6.2 Differentiate Between Concurrency And Parallelism

6.2.1 Concurrency And Parallelism

Concurrency and parallelism are often understood as synonyms – which they are not.

Concurrency means that an application is able to execute multiple tasks at once where the order in which they are executed is not predetermined. This is true for multithreaded applications. But also for applications which use only one thread to handle concurrent tasks, by using techniques like actors, event loops or coroutines.

Contrary to concurrency, parallelism means that two tasks are executed in parallel on different CPUs. This requires two things: An application running on different threads or processes and a computer equipped with multiple CPUs or a CPU with multiple cores.

6.2.2 Concurrency And Parallelism In Realtime Web Applications

As described before, concurrency and parallelism are different things. One is not a subset of each other, but they overlap. The point in which they overlap is the most interesting part for realtime web applications: Serving WebSocket or Comet clients requires most importantly high concurrency. But to achieve really high concurrency on a single machine, all resources need to be utilized and therefore it is necessary that the application serving these concurrent clients also runs in parallel.

Threads and processes are essentially constructs which allow parallel programming but also enable concurrent programming. The problem is that exploiting threads or processes to gain concurrency leads to an application which is automatically parallel but leads also to an application which wastes lots of resources. This is not necessary for applications which only need high concurrency.

Using more lightweight solutions like actors, coroutines or event loops which are designed primary to build concurrent programs leads to less overhead and therefore to applications able to serve multiple concurrent connections without wasting resources. Combining these lightweight mechanisms with threads or processes to utilize all available CPUs leads to much better resource utilization: These applications are both, concurrent and parallel, but use parallelism only to gain higher concurrency.

6.3 The Future Will Be Async

WebSockets and Comet connections all show a similar traffic profile: Long living TCP/IP connections with small amounts of traffic and a need for low latency. Additionally a realtime web application should be able to support big amounts of users. The software needed to power these systems on the server side is heavily I/O-bound. Having heavyweight solutions with scheduling and memory overhead which increases linearly with the amount of users is therefore not the best way to handle these connections.

The old fashioned way of handling concurrent connections by forking of new processes to handle each connection in a dedicated process is certainly too heavyweight and wastes too much resources.

Using threads to provide shared memory concurrency is a bit more lightweight but introduces new problems caused by race conditions and deadlocks.

The currently most promising techniques to support WebSockets and Comet are the actor model and asynchronous communication backed by an event loop. Both systems use asynchronous communication: Actors communicate asynchronously whereas an event loop normally uses asynchronous I/O.

Depending on the implementation, actors are able to scale on different CPUs and even different machines. A downside of this system is that it adds complexity to a system and is not supported by all programming languages.

An event loop on the other side can handle a lot of connections in a really efficient way. There is no need to special support of the runtime and there are frameworks available on most general purpose programming languages. Scaling such an application to more CPUs or machines on the other hand is much harder than scaling an actor system with built in remote-actor support.

6.4 Use The Right Tool For The Job

A recommendation which applies to a lot of different topics also applies in the case of realtime web applications: Use the right tool for the job.

Processes and kernel threads are primitives an operating system is able to schedule and therefore tools which can be used by virtual machines and other runtime environments. For web applications they are clearly too heavyweight and come with too much initialization and scheduling cost. Using more lightweight tools like coroutines, actors or event loops is definitely the way to go. To decide which model to use, a developer needs to be able to estimate how much users the application should be able to support and in which environment it will be deployed.

6.5 A Need For Better Abstractions To Model Concurrency

Mechanisms for modelling concurrency in an efficient way are available for some time: The papers describing actors, coroutines or event loops were written decades ago. However the wasteful model of serving a connection with a dedicated process or thread is still one of the most dominant despite all the related problems. One reason for this is that

these abstractions mostly add complexity to the programming model, are harder to debug and therefore harder to reason about. Better abstractions for managing concurrent applications are therefore needed to create applications which are both, maintainable and able to serve thousands of clients per machine.

Bibliography

- AB, E. (2010). Erlang documentation: Processes. http://www.erlang.org/doc/efficiency_guide/processes.html.
- Alexander Libman, V. G. (2005). Comparing two high-performance i/o design patterns. http://www.artima.com/articles/io_design_patterns.html.
- Ali, M. (2008). Ruby fibers vs ruby threads. <http://oldmoe.blogspot.com/2008/08/ruby-fibers-vs-ruby-threads.html>.
- Armstrong, J. (2007). *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf.
- Ben-Yosef, A. (2011). You owe it to yourself to be old-school. <http://www.codelord.net/2011/02/22/you-owe-it-to-yourself-to-be-old-school/>.
- Conway, M. E. (1963). Design of a separable transition-diagram compiler. *Commun. ACM*, 6:396–408.
- Coplien, E. J., Schmidt, D. C., and Schmidt, D. C. (1995). Reactor - an object behavioral pattern for demultiplexing and dispatching handles for synchronous events. <http://www.cs.wustl.edu/~schmidt/PDF/Reactor.pdf>.
- de Moura, A. L. and Ierusalimschy, R. (2004). Revisiting coroutines. Technical report. <http://www.inf.puc-rio.br/~roberto/docs/MCC15-04.pdf>.
- Duarte, G. (2008). What your computer does while you wait. <http://duartes.org/gustavo/blog/post/what-your-computer-does-while-you-wait>.
- Eric A. Meyer, B. B. (2001). Introduction to css3. Technical report, W3C. <http://www.w3.org/TR/css3-roadmap/>.
- Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Rfc 2616, hypertext transfer protocol – http/1.1.

- Garrett, J. J. (2005). Ajax: A new approach to web applications. <http://www.adaptivepath.com/ideas/essays/archives/000385.php>.
- Google (2010). V8 javascript engine: Design elements. <http://code.google.com/apis/v8/design.html>.
- Graham, P. (2004). *Hackers and Painters: Big Ideas from the Computer Age*. O'Reilly.
- Grigorik, I. (2008). Concurrency is a myth in ruby. <http://www.igvita.com/2008/11/13/concurrency-is-a-myth-in-ruby/>.
- Grigorik, I. (2009). Fibers & cooperative scheduling in ruby. <http://www.igvita.com/2009/05/13/fibers-cooperative-scheduling-in-ruby/>.
- Grigorik, I. (2010). Concurrency with actors, goroutines & ruby. <http://www.igvita.com/2010/12/02/concurrency-with-actors-goroutines-ruby/>.
- Group, T. O. (1997). The single unix specification, version 2. <http://pubs.opengroup.org/onlinepubs/007908799/xsh/realtime.html>.
- Heilmann, C. (2010). Websocket disabled in firefox 4. <http://hacks.mozilla.org/2010/12/websockets-disabled-in-firefox-4/>.
- Hickey, R. (2010). Values and change - clojure's approach to identity and state. <http://clojure.org/state>.
- Hickson, I. (2010). The websocket api. Technical report, W3C. <http://dev.w3.org/html5/websockets/>.
- Hickson, I. and Hyatt, D. (2010). HTML 5: A vocabulary and associated APIs for HTML and XHTML. Technical report, W3C. <http://dev.w3.org/html5/spec/>.
- Hoare, C. A. R. (1983). Communicating sequential processes. *Commun. ACM*, 26:100–106.
- Holdener, III, A. T. (2008). *Ajax: the definitive guide*. O'Reilly, first edition.
- Jäger, K. (2008). Finding parallelism - how to survive in a multi-core world. Technical report, HdM Stuttgart. <http://kaijaeger.com/finding-parallelism-how-to-survive-in-a-multi-core-world.html>.
- Kegel, D. (2006). The C10K problem. Technical report, Kegel.com. <http://www.kegel.com/c10k.html>.

- LearnBoost (2010). socket.io: Sockets for the rest of us. <http://socket.io>.
- Lee, E. A. (2006). The problem with threads. *Computer*, 39:33–42. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf>.
- Lua-Documentation (2010). Lua documentation. <http://www.lua.org/manual/5.1/manual.html#2.11>.
- Lundin, K. (2008). Inside the erlang vm - with focus on smp. Technical report, Erlang User Conference. http://ftp.sunet.se/pub/lang/erlang/euc/08/euc_smp.pdf.
- Mailinglist, N. (2011). I love async, but i can't code like this. http://groups.google.com/group/nodejs/browse_thread/thread/c334947643c80968/8d9ab9481199c5d8.
- Manpage (2008a). Linux man page: epoll. <http://linux.die.net/man/4/epoll>.
- Manpage (2008b). Linux man page: fork. <http://linux.die.net/man/2/fork>.
- Manpage (2008c). Linux man page: poll. <http://linux.die.net/man/2/poll>.
- Manpage (2008d). Linux man page: select. <http://linux.die.net/man/2/select>.
- Manpage, L. (2008e). Linux programmer's manual: pthread_create - create a new thread. http://www.kernel.org/doc/man-pages/online/pages/man3/pthread_create.3.html.
- MDN (2010). Same origin policy for javascript. https://developer.mozilla.org/en/Same_origin_policy_for_JavaScript.
- Moura, A. L. D., Rodriguez, N., and Ierusalimschy, R. (2004). Coroutines in lua. *Journal of Universal Computer Science*, 10:925. <http://www.inf.puc-rio.br/~roberto/docs/corosblp.pdf>.
- Oracle (2010). Java threading documentation. <http://www.oracle.com/technetwork/java/threads-140302.html>.
- Philipp Haller, M. O. (2007). Actors that unify threads and events. Technical report, École Polytechnique Fédérale de Lausanne (EPFL). <http://lamp.epfl.ch/~phaller/doc/haller07actorsunify.pdf>.

- Pyarali, I., Harrison, T., Schmidt, D. C., and Jordan, T. D. (1997). Proactor - an object behavioral pattern for demultiplexing and dispatching handlers for asynchronous events. Technical report, Washington University. <http://www.cs.wustl.edu/~schmidt/PDF/proactor.pdf>.
- Ruby-Documentation (2010). Ruby fiber documentation. <http://www.ruby-doc.org/core-1.9/classes/Fiber.html>.
- Shaw, Z. A. (2010). poll, epoll, science, and superpoll. <http://sheddingbikes.com/posts/1280829388.html>.
- Tomayko, R. (2009). I like unicorn because it's unix. <http://tomayko.com/writings/unicorn-is-unix>.
- Ubl, M. (2010). Introducing websockets: Bringing sockets to the web. <http://www.html5rocks.com/tutorials/websockets/basics/>.
- Welsh, M. D. (2002). An architecture for highly concurrent, well-conditioned internet services. Technical report. <http://www.eecs.harvard.edu/~mdw/papers/mdw-phdthesis.pdf>.

7 Appendix

While doing my research I was wondering what professionals think about the topic. I wrote an email to some people who are working on interesting projects related to my thesis and asked for their opinion.

”Web applications are very popular these days. Especially realtime applications like multi-player games and software for realtime communication and collaboration are getting a lot of attention. Comet has been out there for years and with the emergence of WebSockets the trend of web applications with realtime features will continue. Scaling these application on the server side to support hundreds and thousands of users is an interesting challenge:

- The simple request/response model does not apply anymore - Long living stateful TCP connections are needed to support realtime services

The challenge to scale these services, both vertically and horizontally, is the main topic of my thesis. The thesis is almost finished and now I’m interested what professionals think about this topic.

What are the most interesting techniques and patterns to support a lot of users in your opinion? I’m covering processes, threads, the actor model, event loop systems an co-routines in my thesis. What do you think about these approaches? Do you know some other interesting models that weren’t on my radar so far?”

7.1 David Pollack, Creator Of the Lift Web Framework

”In general, Actors lend themselves best to the event-based model that will support many simultaneous users with open connections, but not a lot of work going on. Basically, the event based model where an event leads to computation seems to be the best.

I think people confuse the transport (HTTP) with the computation model (events). I don't think HTTP changes the model... or that web sockets changes the model... it's just events that change state and responses to the changed state.

See <http://www.quora.com/David-Pollak/Comet-programming/answers>"

7.2 Ilya Grigorik, Technology Blogger and Founder Of Postrank

"Hmm, well that's a loaded question ;-) .. Let me try to unpack it a bit:

Many of the conversations I see about "realtime" lack context. First off, when we say "realtime" on the web, we mean "soft realtime", which also means that depending on the application in question, the implementation can vary quite a bit in terms of technology, latency, and so forth.

Fun piece of trivia: early versions of facebook's wall aggregated activity from your friends and randomly distributed their updates into near future, such that when you reloaded the page it seemed like the activity was coming in "in real-time".. when really, it was lagging behind. Lesson: perception of real-time also matters, not just the hard delivery / technology platform.

If you look at large telecoms / SMS carriers, then you'll definitely find that the actor model is pretty popular there. Arguably, that's a sign that those approaches work well for that kind of workloads.

I've spent most of my time with the evented model, in part because I find it easier to model the asynchronous communication with it. Don't get me wrong, nothing against threads:

<http://www.igvita.com/2010/08/18/multi-core-threads-message-passing/>

Having said that, I also have a feeling that threads are just not the right primitives to use when we think about concurrency:

<http://www.igvita.com/2010/12/02/concurrency-with-actors-goroutines-ruby/>

You've probably covered this stuff in your thesis, but I would really dig into architecture of some of the mobile carriers such as Blackberry, iPhone, etc

and see how they're implementing their notification systems. Incidentally, this might be interesting:

blog.urbanairship.com/blog/2010/09/29/linux-kernel-tuning-for-c500k/

Nokia recently announced that their phones will support pubsubhubbub delivery of notifications! Exciting stuff...

<http://www.igvita.com/2010/10/05/case-for-smartphone-web-activity-feeds/>

—

Sorry about the rant, it's definitely a topic near and dear to my heart.. having a hard time trying to figure out how to capture even a fraction of it in a single email. :)"

7.3 Roberto Ostinelli, Creator Of The Misultin Framework

keeping open sockets has a file handle cost on the OS, to say the least. it may also have considerable ram requirements in those frameworks still based on OS processes/thread instead of callback/generator or lightweight processes [such as in erlang, but this can also be achieved in other languages]. http is still very used, it is a very well known protocol with a lot of common techniques to get the most out of it. other areas are much more delicate to enter, and these elements need to be considered when developing enterprise systems. don't forget that VHS won over sony betamax, and the latter was a better product.

the real choice is due to how much communication needs to be handled towards an individual external entity. it is still cheaper to have request/response mechanisms when communication is diluted. it only gets interesting to keep open connection when communication needs to be event driven or there is a lot of it.

i don't know which models you are covering. real life tends to be different from the theoretical approaches, and it's always a matter of tradeoff. there never is the best tool ever TM, instead, always use the best tool for the

specific job. supporting 'lots of users', i'm afraid, can be very different if you need concurrency or not.

i think the main hot topic in scalability is understanding what really needs to be scalable, your approach to single point of failure, the CAP theorem in databases; you need to see when you really need infra-process communication [best example is still chat systems],... you are covering quite a vast area and i can only give pointers to you.