University of Ljubljana
Faculty of Computer and Information Science

Jan Varljen

Scalability and High Availability in Real-time Cloud Services

MASTER'S THESIS

THE 2nd CYCLE MASTER'S STUDY PROGRAMME
COMPUTER AND INFORMATION SCIENCE

Supervisor: Assoc. Prof. PhD Mojca Ciglarič

Ljubljana, 2017

Acknowledgments

I would first like to thank my thesis mentor Assoc. Prof. Ph.D. Mojca Ciglarič for steering me in the right direction and helping me finish my thesis. Her guidance helped me to better understand the problems we were researching but also the importance and methods of scientific work.

I would also like to thank my wife, family and all my friends who continued to motivate me during this time. Without their support, I would never have been able to finish my thesis.

Jan Varljen, 2017

Contents

List of used acronmys

acronym	**meaning**
WWW	World Wide Web
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
UA	User agent
RTCE	Real-time collaborative editing
UX	User experience
IETF	Internet Engineering Task Force
W3C	World Wide Web Consortium
RFC	Requests for Comments
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
IP	Internet Protocol
URI	Uniform Resource Identifier
SSL	Secure Socket Layer
HTTPS	Hypertext Transfer Protocol Secure
RIA	Rich Internet Applications
RTMP	Real-Time Messaging Protocol
AJAX	Asynchronous Javascript and XML
XML	Extensible Markup Language
JSON	JavaScript Object Notation
XHR	XMLHttpRequest

DOM	Document Object Model
BOSH	Bidirectional-streams Over Synchronous HTTP
XMPP	Extensible Messaging and Presence Protocol
SSE	Server-sent events
GUID	Globally Unique Identifier
MTBF	Mean Time Between Failures
MTTR	Maximum Time To Repair
SLA	Service-Level Agreement
WSLA	Web Service-Level Agreement
VM	Virtual Machine
OTP	Open Telecom Platform
COC	Convention Over Configuration
DRY	Don't Repeat Yourself
RPM	Requests Per Minute
AWS	Amazon Web Services
DNS	Domain Name System
CDN	Content Delivery Network
EC2	Elastic Compute Cloud

Povzetek

Naslov: Skalabilnost in visoka razpoložljivost oblačnih storitev v realnem času

Namen magistrske naloge je raziskava tehnologij, ki podpirajo komunikacijo v realnem času, v spletnih aplikacijah in njihov vpliv na skalabilnost in razpoložljivost. Predlagali bomo alternativni pristop izboljšave le-tega z uporabo programskega jezika Erlang. V prvem delu raziščemo obstoječe tehnologije za razvoj spletnih aplikacij v realnem času in pojasnimo zahtevke za skalabilnost in visoko razpoložljivost. V drugem delu naloge smo zgradili štiri prototipe strežnik-odjemalec (eng. client-server) in dva strežniška (eng. server) prototipa ter jih testirali skozi več testnih scenarijev. Vse to, z uporabo avtomatskih skript in na distrubuiranih testnih arhitekturah, postavljenih v oblaku. Na podlagi rezultatov lahko zaključimo, da strežniški nabor tehnologij (predvsem programski jezik) znatno vpliva na alokacijo virov in s tem izboljšuje skalabilnost in visoko razpoložljivost končnega produkta.

Ključne besede

spletne storitve v realnem času, WebSocket, Erlang, skalabilnost, visoka razpoložljivost

Abstract

Title: Scalability and High Availability in Real-time Cloud Services

The goal of this thesis was to research technologies that support real-time communication in web applications and, in particular, implications on scalability and high availability. The thesis proposes an alternative approach to improving scalability and high availability by using Erlang, a highly concurrent programming language. The first part of the thesis researches existing technologies used for developing real-time web applications and explains the scalability and high availability requirements. In the second part of the thesis, four client-side prototypes and two server-side prototypes are built and several test scenarios are performed using automated scripts and cloud-based distributed load testing architecture. From the collected results it can be concluded that the server's underlying technology stack, most of all the programming language, can significantly impact the resource allocation and therefore consecutively improve scalability and high availability of the solution.

Keywords

real-time web services, WebSocket, Erlang, scalability, high availability

Chapter 1

Introduction

Real-time components are becoming ubiquitous parts of modern web applications. Chat services like Facebook Messenger or real-time collaborative tools like Google Docs are great examples of such real-time services. The main concept behind these services is the real-time bidirectional communication where data is simultaneously broadcasted to all the participants.

As real-time components are embedded into existing web applications, they share the same technology stack. However, the World Wide Web and its existing supporting protocols were not designed to work in this real-time environment. In a typical web application flow, the user makes a request and gets the response from the service i.e. the service only has to respond when the user requested something. In the real-time scenario, the service has to respond not only when the user requests the data but also when some specific events are triggered. For that purpose existing protocols were modified and new protocols were designed that are more suitable for real-time communication.

Because web applications consist of both client and server parts, the technologies used on the server solutions also have to be suitable for real-time communication. Server solutions have to be able to accept, process and respond to all the incoming requests while remaining fault tolerant. In a chat service example, when a user types a new message, the content of the message has to be simultaneously sent to all the participants in the conversation. Similarly, a real-time document processor has to make a request to the server for every user keydown event and broadcast that change to all other participants. In general, servers behind real-time services have to cope with an increased load compared to web solutions without real-time components and these new requirements need some different solutions in terms of scalability and high availability.

Contemporary scaling solutions are mainly vertical and horizontal scaling where server power or quantity is increased to meet the needs. But if the software solution on the server, that actually handles the request, is slow, memory inefficient and wastes the server resources then scaling possibilities are limited and ineffective. Different programming languages can be used for developing software solutions but some provide better scalability and high availability features. Choosing the right technology for the client solution and building a server solution that is fast and memory optimized increases the scalability possibilities of the web application.

The expected contribution of this thesis is to design and prototype implementations of client and server solutions that are optimized for handling real-time communication. The solution will prove that given the exact same requirements and hardware capabilities, solutions that are faster and have lower CPU and memory consumption are de facto more scalable and thus better suited for real-time web applications. The solutions will be tested, compared and evaluated by measuring speed, latency, memory consumption, throughput, error rate and mean time before failure.

Chapter 2

Methodology

To better understand the requirements of real-time web applications, the historical development of web applications will be presented. From its beginning in 1989 and the invention of World Wide Web, the emergence of Web2.0 in the early 2000s up to today's modern real-time web applications, the requirements have been changing and driving the development of technologies and protocols needed to support them. A complete overview of existing client-side technologies and protocols for web applications will be given with emphasis on facilitating real-time communication. Each protocol will be described and its design analyzed in the context of real-time capabilities, starting from HTTP, plugins, Comet-like techniques like polling, long polling, and streaming, HTTP/2, Server-sent events up to WebSocket.

Furthermore, the impact of real-time web applications on server solutions will be described where server solutions have to be able to handle more load as the number of requests increase. The importance of scalability and high availability of server solutions for handling real-time web application will be analyzed and examples of vertical and horizontal scaling will be given together with the impact of MTBF and MTTR on availability. The thesis will present the idea of using different server-side technologies to optimize memory consumption and therefore increase the scaling possibilities and fault tolerance. A programming language called Erlang is known especially for its high availability, fault tolerance, and distributed scalability. Erlang is a functional programming language developed by Ericsson designed to support massively scalable real-time systems with requirements on high availability like telecoms, banking, instant messaging, etc. The thesis researches Erlang as a viable solution for developing server solutions for real-time web applications.

Four simple web application solutions will be prototyped that are using polling, long polling, streaming, and WebSocket for handling real-time web behaviors. All the solutions will have the same structure and will exhibit the same feature - a simple chat application that enables multiple users to exchange messages in real time. These applications will be used to compare the advantages and disadvantages of mentioned client-side approaches while handling real-time communication. Applications will be tested using the same test scenario where an automated script will be used to simulate real user interaction. By using network traffic analysis tools, measurements like latency and throughput will be recorded. The thesis will conclude which of the tested client-side approaches gives the best results for handling real-time web applications.

Next, two server-side solutions will be prototyped that handle the requests from the chat application. Both solutions will have the same architecture where they receive messages from the client application, process them, and broadcast the same message to all users connected to the chat. One solution will be developed using the Ruby programming language and it presents an average contemporary server solution. The other solution will be developed using the Erlang programming language and it proposes an improved solution that is better scalable and more resilient to errors. Solutions will be tested using distributed load testing where the load on the server will be simulated by generating a large number of requests with specialized tools like Tsung. Each test will gradually increase the number of requests to the server and the number of successful connections will be recorded. Key performance indicators like CPU load, memory load, average response rate, throughput, error rate, MTBF, etc., will be measured for later analysis.

Given the results of the tests, the solutions will be compared and analyzed and conclusions about the impact of speed and memory consumption on scalability and high availability will be made. The solution that is able to handle more request using the same hardware capabilities while remaining fault tolerant will be considered to be more scalable. Furthermore, the solution that shows an improved MTBF will be considered to have a better high availability factor. The final outcome of the thesis is to support the idea that, besides vertical and horizontal scalability, an alternative approach to scaling real-time web application is by optimizing the solution's resource allocation and therefore effectively improving scalability and high availability.

Chapter 3

Web applications technology overview

3.1 Origins of modern web applications

The most basic definition of a web application is a software architecture consisted of two essential parts: a client and a server that communicate over the World Wide Web. The client is a user agent (UA) responsible for requesting, receiving and displaying data and the server is a computer responsible for serving the data when requested. The user agent is usually a web browser but can also be a web-based robot, command-line tool, mobile app or similar [1]. World Wide Web is often mistakenly taken for the Internet but in fact, the two terms are not synonymous and should not be used interchangeably. The Internet is the world's biggest network infrastructure that connects millions of computers together and gives them the ability to communicate together via a variety of languages known as protocols. Whereas the World Wide Web (abbreviated as WWW or simply the Web) is an information sharing model abstraction built on top of the Internet that uses the Hypertext Transfer Protocol (abbreviated as HTTP) to transmit the data between computers [2]. So the Web is only a portion of the Internet but a very large and important one because most of the traffic on the Internet are actually web applications

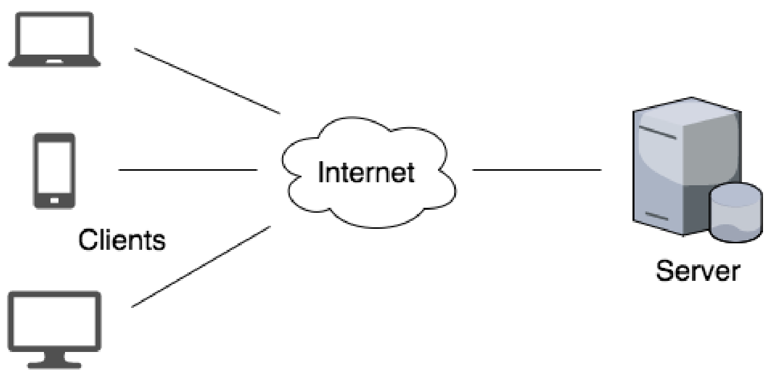


Figure 3.1: The client-server model over the Internet.

communicating over the World Wide Web [3]. The figure 3.1 shows a typical client-server model over the Internet.

The World Wide Web was invented in 1989 by an English scientist Tim Bernes-Lee while employed at CERN in Switzerland. His vision was to build an application layer protocol on top of the Internet for sharing content between computers over the network. When the Web was publicly released in August 1991, this content was just simple static documents formatted with Hypertext Markup Language (abbreviated as HTML) represented by a Uniform Resource Locator (abbreviated as URL) commonly informally termed as web address [4].

In a typical WWW request-response cycle, the client requests a document by requesting the URL through the web browser. The URL pinpoints a server on the network where the resource is located. When this server is found, it responds by sending the requested document back to the client. By receiving the document, the user agent (UA) interprets the data (usually displays the data in the web browser) and the request-response cycle is completed. This typical request-response cycle is shown in the figure 3.2.

Later, additional multimedia content support was added so images, video, audio and similar content could be included in these HTML documents but all in all these documents were simple and primarily designed for information sharing. At that point, the two key actors in the WWW domain were fun-

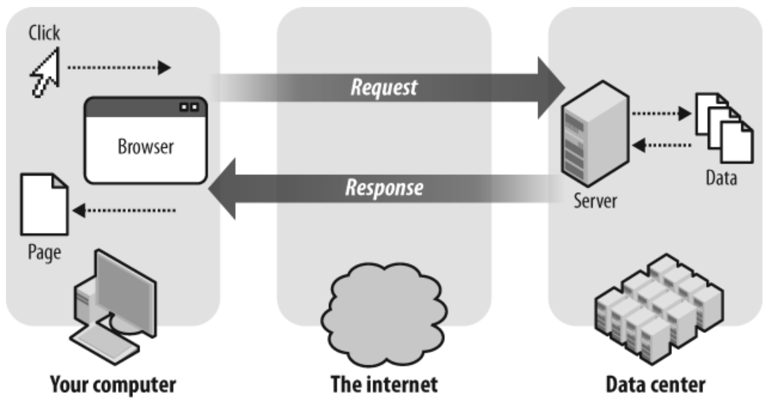


Figure 3.2: A diagram of request-response cycle over the Internet.

damentally separated where content creators were few in numbers and the vast majority of users were simply acting as consumers of content. The most common web pages back then were personal websites hosted on private or free web hosting services such as GeoCities [5]. Today we refer to this first stage of the WWW's evolution with the retronym Web 1.0.

The retronym was introduced after the term Web 2.0 started to appear in the early 2000s when Darcy DiNucci first introduced it in her article ”Fragmented Future” in 1999 [6]. Simple static documents were being replaced by dynamic elements that allow users to interact with the page. With the emergence of Web 2.0, the creator-consumer ratio shifted in favor of creators making them the key actor in this domain. Instead of merely reading a website, a user is invited to contribute to its content by commenting on published articles or creating his own articles. Web pages started to become web applications that provide more than just content and information. They allow users to create content or modify content to their own needs or to share with others. Blogging self-publishing platforms (e.g. WordPress) or social networking websites (e.g. Facebook) attracted millions of users and allowed them to participate in the content creation that drives the modern Web [7] [8] [9].

However, the same request-response cycle was still being used where the client requests and the server respond with the content. While that was a viable solution for the Web 1.0 where the content changes very rarely, in the Web 2.0 every significant change to the web page required a round trip back to the server to refresh the entire page. In modern web applications, the data changes so fast that at the moment the content has been received from the server it can be considered stale because it could have already been changed on the server. The standard unidirectional communication flow where server responds with content only when a client makes a request was not a good enough solution so new tools and technologies were developed to mitigate the emerging problems. These tools and technologies influenced something that is today referred to as the real-time web.

3.2 Real-time web

Real-time web describes modern web applications that enable users to receive information as soon as it is created in the system. To accomplish that, web applications use many different technologies and practices that simulate real-time behavior as closely as possible. The purpose of simulating real-time behavior in web applications is to give users an impression that data exchange is happening instantly - in real time. In reality, this data exchange is never instant and, depending on network lags and server latencies, varies from few milliseconds to few seconds, but if it is fast enough, users have a feeling that it was instant. Responses that are below 0.1 seconds can be considered instant because users perceive it as the system reacted instantaneously [10].

Real-time web applications should not be confused with real-time systems like air traffic control, a car engine or heart pacemakers where specific time constraints have to be guaranteed to prevent total system failures [11]. These systems have its own requirements on real-time computing and this thesis is not interested in researching those systems. In this thesis, the term real-time will be used only in reference to simulating near real-time behavior in web applications.

The key part of the real-time web is to migrate from the unidirectional client-server communication to the bidirectional communication where the server can initiate the communication with the client at any time. This way the client can be notified when the data changes on the server and the server can voluntarily send the data to the client.

Real-time web applications can be separated into two categories:

- Full-fledged real-time applications
- Web applications with real-time components

Real-time collaborative editing (RTCE) applications are a typical example of full-fledged real-time applications. RTCE applications allow multiple users to collaborate on the same document simultaneously. All the changes users are making to the document have to be broadcast in real-time to all the participants in order to allow them to collaborate without conflicts. Compared to traditional nonreal-time collaborative tools where every user has his own version of the document which is later merged into the same document by a version control system, in RTCE everybody is making changes to the same document on the server.

Even though RTCE application existed before the Web 2.0 phenomenon, it was Web 2.0 that caused an explosion of browser-based document editing tools and most RTCE applications today are web-based. Probably the most popular RTCE application today is the Google Docs suite where users collaborate on Docs, Sheets or Slides in real time [12] [13] [14].

Second real-time web category includes all web applications that are not full-fledged real-time applications but have at least one real-time component. This real-time component can be a customer support chat integrated inside the web application where users can exchange real-time messages with the support team or a notification center that informs users about some events happening in the application also in real time.

Real-time techniques found its application in various fields like collaborative groupware tools [15] [16], real-time group communications tools [17] [18] [19], monitoring applications [20], video streaming [21], remote robot teleoperation [22], multi-player online gaming [23], e-learning platforms [24] etc. A lot of different tools and techniques have been used in the past and present to simulate real-time behavior in web applications but the thing they all have in common is the foundations in the HTTP protocol that also powers the World Wide Web.

3.3 HTTP

The Hypertext Transfer Protocol (HTTP) was developed together with the World Wide Web by Tim Bernes-Lee at CERN. It was designed as a stateless application layer protocol that supports the data communication for the WWW. The first version of the HTTP protocol (HTTP v0.9) was published in 1991 defining basic principles of a client requesting a page from a server [25]. The client sends a request containing a special word "GET" together with the document address and the server responds with a message in Hypertext Markup Language (HTML). This basic HTTP GET request and response cycle is shown in the figure 3.3.

After this initial version, further development of HTTP was coordinated by the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C). They continued to improve the protocol with a series of publications called Requests for Comments (RFCs). Through these publications, computer scientists and engineers are discussing methods, behaviors, research, and innovations that could be integrated into a standard. This way the HTTP protocol was expanded and operations, extended negotiation, richer meta-information, security concerns and header fields were added in the RFC 1945 which officially recognized HTTP V1.0 in 1996 [26]. The following version of the protocol (called HTTP/1.1) was officially released in January 1997 in the RFC 2068 [27] but reissued again in June 1999 with

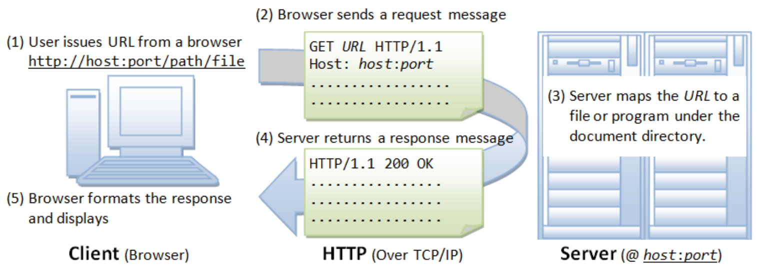

(2) Browser sends a request message
(1) User issues URL from a browser
http://host:port/path/file
GET URL HTTP/1.1
Host: host:port
(3) Server maps the URL to a file or program under the document directory.
(4) Server returns a response message
HTTP/1.1 200 OK
(5) Browser formats the response and displays
Client (Browser)
HTTP (Over TCP/IP)
Server (@ host:port)

Figure 3.3: Basic HTTP GET request and response cycle.

improvements and updates in the RFC 2616 [28]. The final version of the HTTP/1.1 protocol was released in 2014 in the RFC 7230 that revised and clarified some parts of the protocol like caching, authentication, range requests, conditional requests, message syntax and routing [29].

HTTP was designed as an application layer protocol that works on top of existing transfer layer protocols in the Internet protocol suite. The Internet protocol defines four abstract layers which provide end-to-end data communication between computers on the Internet. The first layer is the link layer which defines communication methods for inside a single network segment. The second one is the internet layer which connects independent networks, and therefore, provides internetworking. On top of that, there is the transport layer handling host-to-host communication and the application layer which provides data exchange for applications. The transfer layer protocol underlying HTTP can be either a reliable delivery protocol such as Transmission Control Protocol (TCP) [30] or an unreliable delivery protocol such as User Datagram Protocol (UDP) [31]. Most major Internet applications such as the WWW, email, file transfers require reliable delivery and relies on TCP so the entire suite is commonly referred to as TCP/IP meaning TCP over the Internet protocol (IP) [32].

The HTTP protocol defines the requirements for the request/response communication and sets rules for client and server behaviors. A client sends an HTTP request to a server in form of a request message containing three parts: request-line, header fields, and a message body. The request-line includes a method, Uniform Resource Identifier (URI) [33], and protocol version. The header fields can contain various metadata such as request modifiers, client information, or application specific data. The message body can include any arbitrary data and represents the payload of the request but can also be empty, or even nonexistent in some specific request methods. A server responds by sending one or more HTTP response messages, each containing a status-line, header fields, and a message body. The status-line includes a protocol version, success or error code, and textual reason phrase. The format of header fields is the same as in the request and can include various metadata about the server response. The message body contains the payload of the response but can, the same as for the request, be empty or nonexistent.

The easiest way to test the request/response format is by using the `curl` command in the terminal [34]. The following example shows format and content of data exchange for a GET request generated by running the command `curl -v http://www.example.com/index.html` in the terminal.

The client request:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: curl/7.51.0
Accept: */*
```

In the request, we can identify the request-line with the method (`GET`), the resource identifier (`/index.html`) and the protocol version (`HTTP/1.1`), followed by the set of key-value header fields. There is no message body because the GET method does not have it by the specifications.

The server response:

```
HTTP/1.1 200 OK
```

```
Cache-Control:  max-age=604800
Content-Type:  text/html
Date:  Sat, 11 Mar 2017 15:18:28 GMT
Etag:  "359670651+ident"
Expires:  Sat, 18 Mar 2017 15:18:28 GMT
Last-Modified:  Fri, 09 Aug 2013 23:54:35 GMT
Server:  ECS (iad/18F0)
Vary:  Accept-Encoding
X-Cache:  HIT
Content-Length:  1270

<!doctype html>
<html>
<head>
<title>Example Domain</title>
...
</html>
```

In the response we can identify the status-line with the protocol version (`HTTP/1.1`), success code (`200`) and textual reason phrase (`OK`), followed by the set of key-value header fields. The payload includes the message body and starts after the new line break that indicates the end of the header section. The payload in this example is truncated for brevity but it represents the content of the `index.html` resource.

The HTTP protocol is the foundation of the World Wide Web and web applications. But it was not designed to support real-time web applications so different tools and techniques built around HTTP have to be used to simulate real-time web behavior in web applications.

3.4 Plugins

The first real-time web applications have been the build using Adobe Flash and Silverlight technologies. Adobe Flash is a multimedia software platform used in desktop, mobile and web application developed by Adobe Systems (previously by Macromedia) and Silverlight is a platform for writing and running Rich Internet Applications (RIA) developed by Microsoft. These technologies are used for developing animations, mobile games, desktop application and variety of RIA which also include real-time applications. To support real-time communications, both technologies designed its own proprietary communication protocols that are used instead of the existing HTTP protocol. For example, Adobe Flash designed the Real-Time Messaging Protocol (RTMP) for high-performance transmission of audio, video and data [35].

Both these technologies require to be installed as an external dependency in the browser (as a browser plugin) to run web applications with Flash or Silverlight content and not every browser supports these plugins. This means that users that have incompatible browsers are not able to use web applications and have to install a different web browser. Those who have a compatible browser still have to install external plugins before they can use Flash or Silverlight web application. This makes them more cumbersome to use than native browser functionalities that are available by default in most web browsers. Furthermore, both technologies require a specialized server software solution to handle communication over these proprietary protocols which means standard web servers are not compatible with these technologies and custom solutions like Flash Media Server have to be used [36].

Development of Microsoft Silverlight was discontinued in 2013 mostly because the technology had low adoption rate and was forced out of the market by Adobe Flash. With the emergence of the newest HTML5 standard around 2014 which enables audio and video support as a native browser functionality, even Adobe Flash started to lose its market share rapidly [37]. Recently, mobile platforms dropped the support for Adobe Flash plugins in favor of HTML5 so it is likely that it will also be discontinued at one point.

3.5 Comet

The opposite approach to plugins is to use already existing technologies and native browser functionalities to achieve the same goal - simulate real-time behavior. Comet is an umbrella term encompassing multiple technologies for achieving this interaction. All these methods rely on features included by default in browsers, such as JavaScript, rather than on non-default plugins. Also, they differ from the original model of the web in which a browser requests a complete web page at a time and introduce polling, long polling, streaming, server push and similar approaches for simulating real-time behavior.

3.5.1 Polling

The most basic technique used to simulate real-time behavior is called polling. The idea of this technique is to send a request to the server at given time intervals to check if any changes happened to the resource. This way, when a change happens on the resource, the client receives the update on the next poll request which, in some way, simulates a real-time behavior. The more frequent the poll request, the better real-time behavior is simulated.

Figure 3.4 shows how polling technique generates requests to the server in polling intervals. The requests that have no changes to report back return empty and the one that happened after the event was triggered is returning the actual data.

Polling can be implemented in various ways but the most common one is probably using JavaScript programming language which is included by default in all the web browsers. This technique of retrieving data asynchronously from the server is called Asynchronous Javascript and XML (abbreviated as AJAX). XML stands for Extensible Markup Language and defines a set of rules for encoding document in a format that is both human-

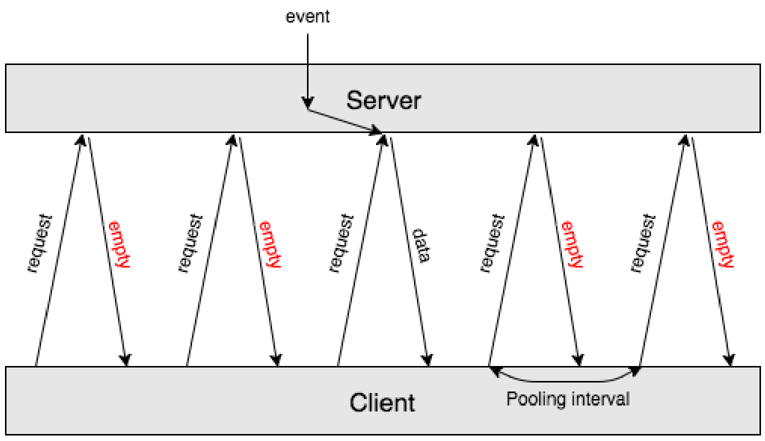


Figure 3.4: Diagram of the polling technique.

readable and machine-readable [38]. XML was used as the standard response format in AJAX request but was later replaced with new standards like JSON. JavaScript Object Notation (abbreviated as JSON) is an open-standard format that uses human-readable text to transmit data object consisted of attribute-value pairs [39]. Some of the advantages of JSON over XML are JSON being native to JavaScript which makes it easier to manipulate with and more human-readable with simpler syntax and less markup overhead. AJAX uses XMLHttpRequest (XHR) object provided by JavaScript for exchanging data asynchronously between browser and server [40].

```
// Initialize the HTTP request.
var xhr = new XMLHttpRequest();
xhr.open('get', 'http://www.example.com/posts/11');

// Track the state changes of the request.
xhr.onreadystatechange = function () {
    var DONE = 4; // readyState 4 means the request is done.
    var OK = 200; // status 200 is a successful return.
    if (xhr.readyState === DONE) {
        if (xhr.status === OK) {
            console.log(xhr.responseText); // This is the returned text.
        } else {
            console.log('Error: ' + xhr.status); // An error occurred during the request.
        }
    }
};

// No data needs to be sent along with the request.
xhr.send(null);
```

In the previous code example an asynchronous HTTP GET request is made to the `www.example.com` host and `posts/11` resource. When the request is done, it checks the status and logs the response text or error code to the browser console. Purpose of this code snippet is to show how easily an asynchronous request to the server using AJAX can be created. Similarly, timer functions can be created that poll a resource every few seconds and return changes, therefore, implement the polling technique.

The biggest advantage of the polling technique is that it only uses already existing technologies like AJAX and XMLHttpRequest that are supported in all the modern web browsers [41]. It uses the existing HTTP protocol and the server solution only has to differentiate the asynchronous AJAX request from the standard GET request and return the JSON representation of resources instead of the full HTML page.

Although this solution seems to be a viable candidate for designing real-time web applications, it has many drawbacks. One of the issues is the asynchronous nature of AJAX and callback driven programming where because of external conditions like network latency, the sequence of the responses can be out of sync with the sequence of the request so additional code behaviors have to taken into account. This can lead to more complex code that is often harder to maintain, debug and to test. But the biggest disadvantage of the polling technique is the increased traffic and load on the servers it generates. Depending on the polling interval every user generates an increased number of requests to the server and all these requests have to go through the network and be handled by the server. Also, only a small amount of these requests actually return something whereas all the rest are just returning empty payloads.

3.5.2 Long polling

To overcome this biggest drawback of polling, long polling technique was introduced. In long polling, the client also sends a request to the server to check for any changes but the server holds the response until the information is available. Once available, the server responds and sends the new information back to the client. When the client receives the new information, it immediately sends another request and this way the operation is repeated. Long polling can be considered as an extended version of polling where the client still has to ask the information every time but the server will reply only when the information is available. This effectively emulates a server push feature and eliminates the huge amount of responses which are only returning the info that no new data is available.

Figure 3.5 shows how long polling technique holds the responses until changes are available. The responses are sent to the client only after an event was triggered on the server.

On the client, long polling can be implemented in several ways and the most straightforward are by using the same XHR technique mentioned before. The browser makes an asynchronous request to the server, which waits for the data to be available before responding. When the client receives and processes the response it immediately sends another XHR, to await the next event. Thus the browser always keeps a request outstanding with the server, to be answered as each event occurs. Compared to the polling example before, the client implementation is exactly the same, a simple AJAX request to the server that waits for the response by checking the `onreadystatechange` status. Only the server implementation has to be different, where it does not respond immediately but waits for the data to be available.

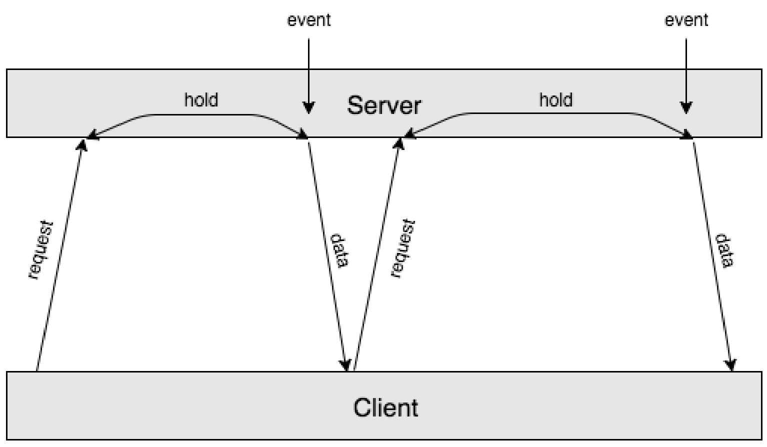


Figure 3.5: Diagram of the long polling technique.

Another way to implement long polling is by using the HTML script tag element. It is used when long polling has to be made to work across different subdomains. Because of Same-Origin Policy enforced by all modern browsers, XHR calls can only be made between web pages with the same origin where the origin is defined as a combination of URI, hostname and port number [42]. So a web page on *http://www.example.com/page.html* can send an XHR request to the same domain (e.g. *http://www.example.com/another-page.html*) but will be blocked to send a request between subdomains (e.g. *http://api.example.com/page.html*). The same way any modifications of the URI, hostname or port number will be blocked by the browser and XHR requests will fail to be sent. The HTML script tag, on the other hand, can be pointed at any URI without the browser blocking the request, and the response will be executed in the current HTML document. So instead of doing an XHR request, an HTML script tag is inserted into the HTML and it makes a request to the server. The server waits for the data the same way as before and responds when the data is available. When the client receives and processes the response it immediately creates another HTML script tag which then makes the request and this way the browser always keeps a request outstanding with the server.

Compared to polling, the biggest advantage of long polling technique is it minimizes the number of requests the client sends to the server thus decreasing the impact on traffic and throughput. The issue with the long polling technique is that the server has to allocate its resources for every outstanding connection held open and has to hold it allocated until it is fulfilled. This makes the technique poorly scalable because server resource can easily get depleted when a large number of clients are connected simultaneously.

3.5.3 Streaming

Both polling and long polling are using the most straightforward HTTP request-response flow where the connection between the client and the server is terminated after the response is received. This is the default and normal behavior of the HTTP protocol because it was designed to work that way. The client opens a connection to the server, requests a page, the server responds with the page, the client receives the page and the connection is terminated. In the polling and long polling scenarios, the client always keeps a request outstanding with the server by initiating a new request immediately after receiving the response. That means terminating existing and opening new connections with the same server repeatedly all the time which negatively impacts the traffic and increases the load on the server.

To overcome this problem, HTTP streaming can be used where a single request is kept open indefinitely and data is streamed over that same request. The request is never terminated and the connection is never closed so the server can push the data back to the client using a single long-living open request. The HTTP streaming mechanism is based on the capability of the server to send several pieces of information in the same response without terminating the request or the connection and both HTTP/1.1 and the older HTTP/1.0 protocols support streaming. When events are triggered on the server the data is streamed to the client in chunks using the same outstanding connection. Only when the connection timeouts or terminates because of an external cause, the new connection is established.

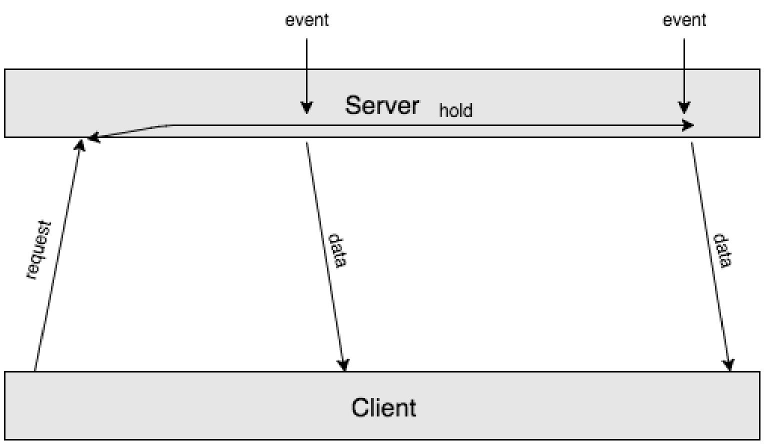


Figure 3.6: Diagram of the streaming technique.

Figure 3.6 shows how streaming technique streams data over a single open connection.

HTTP streaming can be implemented in several ways and the most straightforward are by using a hidden iframe HTML element. The iframe was designed to allow embedding one HTML document inside another and breaking websites into chunked blocks but can also be used for HTTP streaming. An invisible iframe is embedded into the website and it opens a long-living connection with the server called a forever frame. The server sends responses in the form of script tags containing JavaScript code that gets inserted into the hidden iframe. Because the browser renders HTML pages incrementally, each script tag is executed as it is received. This way the website can be updated with the changes from the server e.g. the user can get notified when a new comment is added. The biggest benefit of the iframe method is that because it uses only standard HTML elements like script tag and iframe, it works in every common browser. The downsides of this technique are the lack of reliable error handling and the impossibility of tracking the state of the request process happening inside the iframe.

The biggest advantage of using HTTP streaming is the reduced number of generated requests compared to polling and long polling. Reduced number of requests means a decrease in network traffic and server load. Furthermore, as server responses are just chunks of data inside the same connection and not full HTTP responses, the response headers are omitted so the size of the response is also smaller. In general, this mechanism significantly reduces the network latency because the client and the server do not need to open and close connection for every request. On the other hand, HTTP streaming introduces a lot of new issues that have to be considered. HTTP streaming will not work if any network intermediary between the client and the server, like proxy or gateway, buffer the entire response before forwarding the data. It is a common behavior for some network intermediaries like caching transparent proxies and it is exactly the opposite of how streaming works. Furthermore, one single long-living request can theoretically be used for an endless data stream from the server to the client. In practice, browsers have to timeout and terminate requests occasionally to avoid unlimited growth of memory usage caused by JavaScript and DOM (Document Object Model) elements created in the process of streaming. Similarly, the server also has to occasionally terminate the request to free up allocated server resources like memory and CPU. In both cases, the connection has to be reestablished so the client sends a new request to the server to open streaming. The default timeout value in most modern browsers is 300 second but as some network intermediaries can potentially have shorter timeouts, a 30-second timeout is generally considered to be a safe value [43].

3.5.4 Reverse HTTP

Reverse HTTP is an experimental protocol where bidirectional communication between the client and the server can be achieved by using two HTTP connection. The first HTTP connection is a standard connection between the client and the server and the second one is a reversed connection between the server and the client. The basic concept is to reverse the roles of the client and the server and allow clients to behave as a server in the reversed connection. The client can request to upgrade to a reverse HTTP and then the server can initiate a connection with the client. Now, as events occur on the server, it can directly send them to the client.

Reverse HTTP takes advantage of the HTTP/1.1 Upgrade header to turn one HTTP connection around. A client makes a request to a server with the Upgrade: PTTH/1.0 header and if the server accepts the upgrade the server starts using the connection as a client, and the client starts using the connection as a server. If the server is not able to perform the upgrade, the communication usually fallbacks to a Comet-like technique like long polling [44] [45] [46].

The client request:

```
POST /queue HTTP/1.1
Host:  www.example.com
Upgrade:  PTTH/1.0
Connection:  Upgrade
```

The client request is a POST method to the resource `/queue` on the host server *www.example.com* requesting a protocol upgrade to PTTH/1.0 through the `Update` header.

The server response:

```
HTTP/1.1 101 Switching Protocols
Upgrade:  PTTH/1.0
Connection:  Upgrade
Date:  Sat, 11 Mar 2017 15:18:28 GMT
Content-Type:  text/plain
Content-Length:  0
```

The server accepted the upgrade and informs the client by responding with the 101 Switching protocol status. Immediately after that, the server makes an HTTP request pointing back to the client.

The server request:

```
GET /status HTTP/1.1
Host: 127.0.0.1:65331
Accept: */*
```

The client response:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 12

Hello world
```

The client responded with OK status and an arbitrary body content. The server can now continue to send data to the client by making its own HTTP requests.

The Reverse HTTP protocol allows client-server bidirectional communication and simulates real-time behavior by sending server pushes to the client as reversed requests using the standard HTTP protocol. But the protocol is only experimental and never really had any real-world applications and documented usages. There are a lot of problems and considerations regarding the reverse HTTP protocol like security issues with man-in-the-middle attacks and problems with firewalls and network intermediaries that block PTTH upgrades because they do not recognize the experimental protocol. Probably the biggest problem with Reverse HTTP is that both client and server implementations have to be developed specifically for the needs because modern browsers and server solutions never adopted the Reverse HTTP protocol so no server or client out-of-the-box solutions are available.

3.5.5 BOSH

BOSH was developed by the XMPP Standards Foundation in 2004 and stands for Bidirectional-streams Over Synchronous HTTP [47]. BOSH employs a combination of HTTP long polling mechanism and multiple synchronous HTTP request-response pairs. With the long polling technique, the response is deferred until it actually has any data to send to the client and as soon as the client receives a response it sends another request ensuring that one long-living connection is always open. In some situations, the client needs to send data to the server while it is waiting for some data to be pushed from an open long poll request. To prevent data from being pipelined behind the long poll request that is on hold, the client can send its outbound data using a second HTTP request. This way BOSH forces the server to respond to the request it has been holding as soon as it receives a new request from the client thus preventing slow pipelining. To ensure that the periods with no requests pending are never too long, BOSH defines the negotiation of an inactivity period value. If the server has no data to send to the client for an agreed amount of time then the holding request will be closed, by responding with no data, and a fresh new client request will be triggered immediately. This way BOSH ensures that if a network connection is broken that both parties will realize that fact within a reasonable amount of time. BOSH was designed to transport any data efficiently and with minimal latency in both directions. Compared to most other bidirectional HTTP-based transport protocols and techniques (like AJAX based solutions), BOSH is significantly more bandwidth-efficient and responsive mainly because it gracefully handles special cases like inactivity, overactivity, pipelining or network disruptions [48].

3.5.6 Bayeux

The Bayeux protocol was developed in 2007 by the Dojo Foundation with the primary purpose to support responsive bidirectional interaction between web clients [49]. With Bayeux, asynchronous messages can be delivered from server to client, client to server and even client-to-client communication is possible where an intermediate server takes on the role to moderate the communication. In order to achieve bidirectional communications, a Bayeux client uses a combination of two HTTP connections (similar to Reversed HTTP), HTTP streaming mechanisms and HTTP long polling. Because the HTTP/1.1 specification recommends that a single client should not maintain more than two open connection with any server, the Bayeux protocol implementation uses only two HTTP requests simultaneously and fallbacks to HTTP streaming and long polling techniques [50]. Asynchronous messages are exchanged between web clients using channels where clients are subscribed to receive published events formatted in JSON encoded messages. During the connection negotiation between client and server, handshake messages are used to exchange connection type, authentication methods and to mutually reveal acceptable bidirectional techniques where the client then select the preferred one from those provided by the server. The idea of the Bayeux protocol is to reduce the complexity of developing Comet-like real-time applications by providing simple mechanisms that solve complex problems like message routing and distribution in real-time bidirectional communication [51].

3.5.7 Server-sent events

Server-sent events (abbreviated as SSE) is a technology that allows browsers to receive automatic updates from a server that pushes the content to the client via an HTTP connection. This technology was first implemented as an experimental feature in the Opera web browser back in 2006. The first working draft specification of SSE was published by the W3C in April 2009 and it was finally standardized as part of the HTML5 specifications in 2015 [52]. The SSE standard defines an API that enables servers to push data to the client over HTTP in the form of DOM events.

Because HTTP streaming is the foundation of data exchange in SSE,

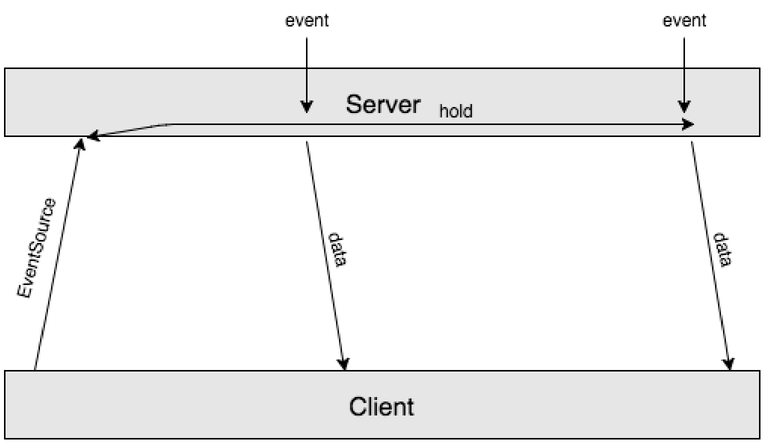


Figure 3.7: Diagram of the SSE technique.

similar behaviors can be detected. Bidirectional communication is achieved by using one long-living request where the server streams the data to the client (shown in figure 3.7). The client connects to the server by opening a stream source request and the server can then stream the data using that open request. The EventSource interface is part of the JavaScript web API and is used to create a long-living connection to a server and receive SSE in `text/event-stream` format [53].

```javascript
// Initialize the stream source.
var source = new EventSource("http://www.example.com/stream")
    ;

source.onopen = function(event) {
  console.log('connection is established');
};

source.onmessage = function(event) {
  console.log(event.data); // This is the returned data.
};

source.onerror = function(event) {
  console.log('an error occurred');
};
```

In the previous code snippet, a new stream source is opened on the URL `http://www.example.com/stream`. On that URL the server accepts requests and starts streaming events. Next, three event handlers `onopen`, `onmessage`, and `onerror` are defined that are triggered when specific events happen. When actual server events are received, the `onmessage` handler is triggered and the event data can be accessed through the `event.data` object.

The benefits of using SSE compared to polling and long polling are similar to HTTP streaming. The number of generated requests is much smaller and therefore the network traffic and server load are decreased. Also, because response headers are omitted in the events being sent from the server, the size of the response is smaller compared to a normal HTTP response. Furthermore, SSE does not have problems with network intermediaries that buffer the response because the `text/event-stream` content type ensures that the response is not buffered or cached by proxy or gateway servers. The `EventSource` provides a clean API for writing simple solutions that do not have issues with memory leaks caused by DOM object manipulations like previous custom iframe HTTP streaming solutions. Most modern browsers have a native support for the `EventSource` interface but some browsers, like the Microsoft Internet Explorer, have never added the support for SSE [54]. This complicates the development of real-time web applications using SSE because various fallbacks have to be implemented to allow users using these browsers to experience the real-time features all the same.

3.5.8 HTTP/2

The HTTP protocol is a wildly successful protocol which forms the foundation of the World Wide Web but can also be found in almost every part of the Internet domain. However, the way HTTP/1.1 uses the underlying transport protocols has several characteristics that have a negative overall effect on application performance today. In particular, HTTP/1.1 is limited to elementary request pipelining and therefore needs to open multiple connections to a server in order to achieve concurrency of the requests. Furthermore,

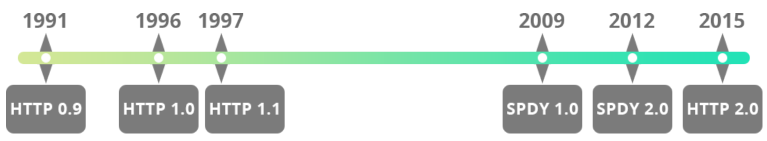


Figure 3.8: Timeline of the HTTP protocol development.

HTTP header fields are often repetitive and verbose, causing unnecessary network traffic as well as causing the transport layer congestion window to quickly fill. These characteristics ultimately result in excessive latency when multiple requests are made on a single TCP connection [55]. Given the fact that the first version of the HTTP protocol was released in 1991, the protocol needed a major update that would improve the shortcomings of the original protocol. The successor HTTP/2 (originally named HTTP/2.0) was standardized in May 2015 in the RFC 7540 and RFC 7541 [56] [57]. Many ideas and concepts found in the HTTP/2 have been influenced by an earlier experimental protocol called SPDY originally developed by Google in 2012. Figure 3.8 shows the timeline of the HTTP protocol development where the relation to the SPDY protocol can be seen.

The primary goal of research and development of the HTTP/2 centers around three concepts - simplicity, high performance, and robustness. It was also really important to maintain high-level compatibility and interoperability with the HTTP/1.1 in terms of methods, status codes, URI and header fields, to facilitate a graceful migration to the new version. Many new concepts were introduced which are essential for decreasing the latency and improving the page load speed in web browsers.

- Data compression of HTTP headers minimizes the protocol overhead and therefore improves the performance with each browser request and server response

- Multiplexing of multiple request streams over a single TCP connection reduces latency and improves performance

- Server Push allows a web server to send resources to a web browser even before the browser gets to request them which significantly improves the performance while loading assets like images or `.js` and `.css` files
- HTTP/2 uses a binary protocol instead of a textual one for processing command in the request-response cycles which is less error-prone and has a lighter network footprint making the protocol more robust
- HTTPS and TLS are required by default which makes the protocol more secure

The HTTP/2 Server Push allows the server to send additional cacheable information to the client that is not requested directly but is anticipated to be requested in future requests. For example, a client requests an HTML page and receives it. While the browser interprets the page content it finds several JavaScript and CSS files to be linked inside. Now the browser has to make additional requests to get the content of those files. With the HTTP/2 Server Push requirement of these files would be anticipated and they would be sent along with the HTML page content inside the same request which brings significant performance improvements. Figure 3.9 shows the benefits of HTTP/2 Server Push compared to HTTP1.1 and HTTP/2 without the push technique. With HTTP1.1 all the resources are downloaded sequentially making the total time to receive the page content 5 seconds. In the second example, the HTTP/2 multiplexing feature enables downloading the assets in parallel making the total time to receive the page content 3 seconds. In the final example, by using the Server Push feature the whole page content is received in only 2 seconds because all the assets are pushed from the server together with the page inside the first request.

The Server Push technology looks like an interesting technique for real-time web applications but it cannot be used to push arbitrary data. It was designed to push only asset files like `.js`, `.css` and images referenced in the HTML document and can be only processed by browser not by applications. Nevertheless, HTTP/2 can be used to improve the performance of

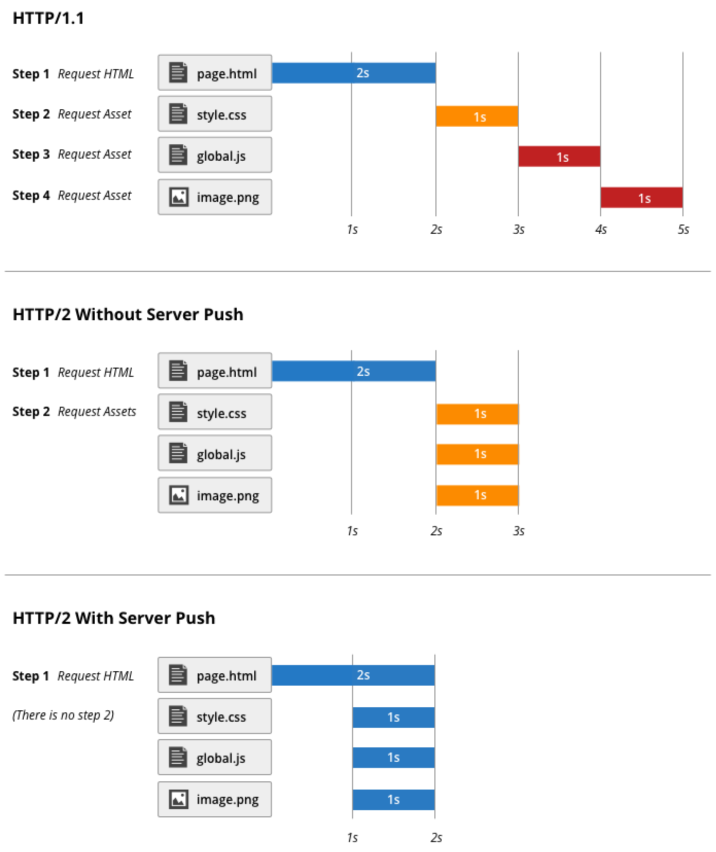


Figure 3.9: Benefits of HTTP/2 Server Push

real-time web applications especially in combinations with SSE. Because of all the benefits already mentioned like header compression, binary protocol, etc., real-time applications with SSE over HTTP/2 are faster compared to HTTP/1.1 applications [58].

3.5.9 Summary

All the technologies and techniques mentioned so far are using the same underlying HTTP protocol for data exchange between the client and the server. Because the HTTP protocol was designed to support only unidirectional client-server communication, these Comet-like technologies are trying to simulate real-time behaviors by forcing bidirectional communication in different ways.

Polling and long polling are using multiple HTTP requests to poll for changes in certain intervals. HTTP streaming is using a single long-living HTTP request to stream the changes back to the client. Reverse HTTP is using two parallel opposite HTTP requests between the client and the server to simulate full-duplex communication. BOSH and Bayeux technologies are using a combination of long polling, HTTP streaming and Reverse HTTP techniques to provide a sustainable solution with graceful fallbacks and improved performance. Similar to HTTP streaming, Server-sent events are also using a long-living HTTP request to push changes back to the client in form of server events.

The next generation of the HTTP protocol, the HTTP/2 protocol, brought many new features and enhancements which improved the overall performance of the protocol but the bidirectional communication is not part of the protocol and therefore Comet-like techniques still have to be used to simulate real-time behaviors.

There was a desperate need for a revolutionary new protocol that would change the way bidirectional communication is handled over the Internet so the IETF group started working on the new protocol called WebSocket.

3.6 WebSocket

WebSocket is a communications protocol that provides full-duplex communication channels over a single TCP connection. The origin for the protocol was the TCP-based socket API called TCPConnection first referenced in the HTML5 specification in 2008 [59]. After that, Ian Hickson and Michael Carter collaborated on enhancing the protocol and ultimately coined the name WebSocket protocol. In December 2009, Google Chrome 4 was the first browser to include a full support for the WebSocket standard, enabling it by default [60]. After the protocol was shipped and enabled by default in multiple browsers, the RFC was finalized and the protocol was finally standardized by the IEFT in 2011 as RFC 6455 [61].

The WebSocket protocol was designed to supersede existing Comet-like bidirectional communication technologies that use HTTP as a transport layer but to reuse the existing infrastructure at the same time. It uses the same HTTP ports 80 and 433 and this way supports existing HTTP proxies and intermediaries and avoids being blocked by the firewall which usually blocks all non-web Internet connections. Besides that, its only relationship to HTTP is that the protocol's handshake is being interpreted by HTTP server as an HTTP Upgrade request. The WebSocket handshake allows the browser to start the connection as an HTTP connection and then to gracefully upgrade to WebSocket if possible. In the case when the server does not support the WebSocket communication, the connection cannot be upgraded and standard HTTP requests will be used. This way a full backward compatibility with the pre-WebSocket world is guaranteed. The specifications defines `ws` and `wss` as two new uniform resource identifiers (URI) schemes. The `ws` (WebSocket) scheme opens an unencrypted connection on the port 80 and the `wss` (WebSocket Secure) scheme opens an encrypted connection over the port 433. Figure 3.10 shows how WebSocket protocol connects and receives events from the server but also sends the data from the client to the server which represents a full-duplex communication.

The protocol has three parts: an opening handshake, the data transfer,

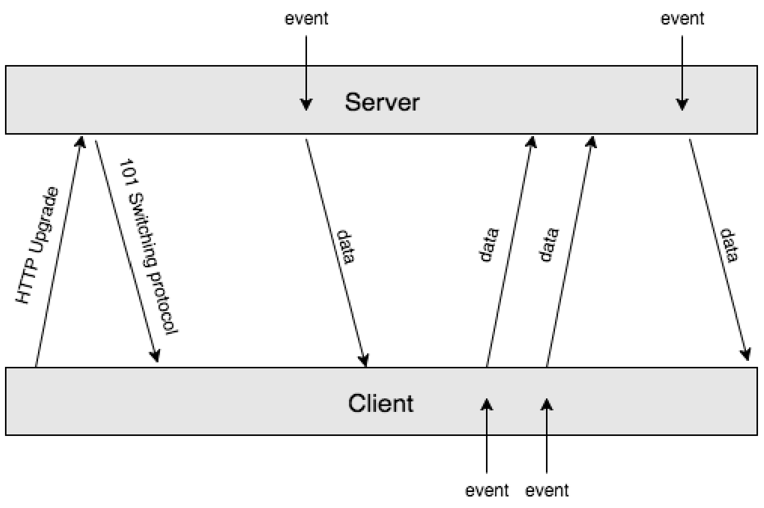


Figure 3.10: Diagram of the Websocket full-duplex communication.

and a closing handshake. The WebSocket handshake request is sent by the client and the server responds with a WebSocket handshake response.

The client handshake request:

```
GET /chat HTTP/1.1
Host:  example.com
Upgrade:  websocket
Connection:  Upgrade
Sec-WebSocket-Key:  dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Protocol:  chat, superchat
Sec-WebSocket-Version:  13
Origin:  http://www.example.com
```

The client handshake request is a GET method to the resource `/chat` on the host server `example.com` requesting a protocol upgrade to WebSocket through the `Update` header. Header fields are used to select different options in the WebSocket protocol.

The Server handshake response:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

The server accepted the upgrade and informs the client by responding with the 101 Switching protocol status. Header fields are used to confirm negotiated WebSocket protocol options.

Once the client and the server have both sent their handshakes, and if the handshake was successful, then the data transfer part starts. The established connection between the client and the server is a two-way communication channel where each side can, independently from the other, send data at will. This data transfer is encapsulated in conceptual units referred to in the specification as `messages`. On the wire, a message is composed of one or more frames and a small header. The data is minimally framed with just two bytes to keep the size of the payload as small as possible.

The closing handshake is far simpler than the opening handshake where either peer can request a closing handshake by sending a control frame with data containing a specific control sequence. Upon receiving such a frame, the other peer sends a Close frame in response and discards any further data received thus closing the connection.

The WebSocket API interface is part of the HTML5 specification and is currently supported in most major browsers including Google Chrome, Microsoft Edge, Internet Explorer, Firefox, Safari and Opera [62] [63]. The client connects to a server by creating a new WebSocket interface to the end-point represented with an URL. The `ws` and `wss` prefixed are used to indicate a WebSocket or a secure WebSocket connection, respectively.

```
// Initialize the secure WebSocket channel.
var socket = new WebSocket("wss://www.example.com/chat");


```

```
socket.onopen = function(event) {
  console.log('connection is established');
};

socket.onmessage = function(event) {
  console.log(event.data); // This is the returned data.
};

socket.onerror = function(event) {
  console.log('an error occurred');
};

socket.onclose = function(event) {
  console.log('connection is closed');
};
```

In the previous code snippet, a new WebSocket channel with the URL `http://www.example.com/chat` is being opened. Next, four event handlers (`onopen`, `onmessage`, `onerror`, and `onclose`) are defined that get triggered when specific events happen. The WebSocket API interface is surprisingly similar to the SSE EventSource API interface introduced before. The biggest difference is that the WebSocket interface can also send data from the client to the server using the same channel as it is a full duplex channel.

```
socket.send('this is a string being sent'); // Sending
   arbitrary data to the server.
```

This code example shows how arbitrary string data can be sent to the server by calling the `send()` method on the socket interface instance object. When all the data transfers are complete, the connection can be terminated by calling the `close()` method on the socket interface instance object. This starts the sequence of closing the channel between the client and the server by sending the closing handshake request.

```
socket.close();
```

The advantages of the WebSocket protocol compared to the usual network traffic over HTTP - is that the WebSocket protocol does not follow the tradi-

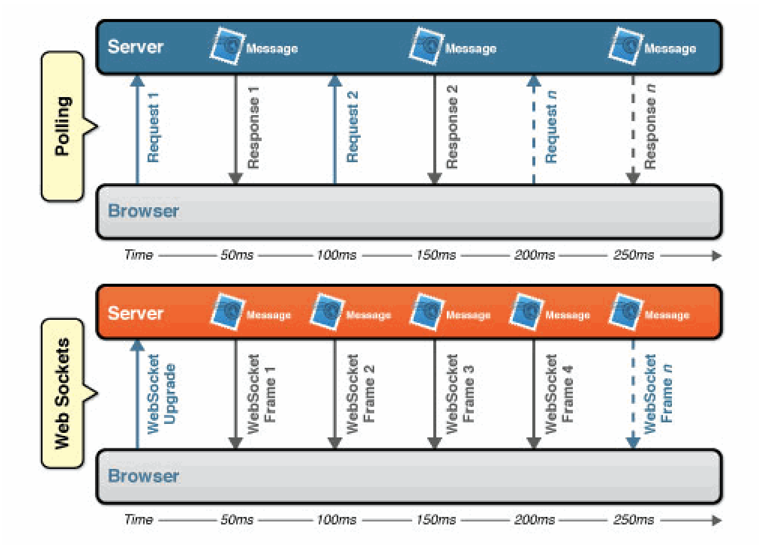


Figure 3.11: HTTP and WebSocket latency comparison diagram.

tional request-response convention. Once a client and a server have opened a channel, both endpoints may asynchronously send data to each other. The connection remains open and active as long as either the client or the server closes the connection. Even if the connection gets closed because of external reasons like network connectivity issues, it can easily be reconnected without much overhead. Compared to previous Comet-like approaches where several round-trips between the client and the server had to be completed before any actual information is sent or received, WebSocket uses the same opened channel for communicating in both directions. Furthermore, the data exchange inside the WebSocket channels is organized into frames which are minimally framed with only a few bytes and do not contain headers. Compared to HTTP message format, WebSocket frames are significantly smaller and that also impact the latency and network traffic [64]. Figure 3.10 shows HTTP and WebSocket protocol latency comparison where given the same amount of time, the WebSocket protocol is able to transfer more data compared to Comet-like approaches like polling.

For the web application to use WebSocket it has to be run in a web browser that supports the WebSocket API. Even though most major browsers today support WebSocket, for those who do not support, fallbacks to traditional Comet-like techniques like pooling have to be implemented. Also, not only the client part of the application has to have the support for WebSocket but the server solution has to be adjusted too. The server solution has to be programmed in an asynchronous programming language that supports handling sockets and broadcasting events. The biggest challenge for the servers is that they have to maintain a large number of open connection to all the clients that are consuming the application. Every open WebSocket channel from each client means a connection on the server that has to be kept alive and processed all the time. Depending on the number of clients, eventually, a server solution will run out of resources which means it has to be scaled to fit the requirements. Scaling and maintaining high availability is an important part of every real-time web application.

Chapter 4

Scalability and high availability

4.1 Scalability

In computer science, scalability is the capability of a system or network to handle growing amount of work by adapting to accommodate that growth. For web applications that means scaling the server solution to be able to handle more load as the number of requests increase. One of the most common ways of scaling computer systems is by adding more resource power. The web application is considered scalable if by adding more resources it can linearly process more requests that before. There are two methods of adding more resources to web applications: horizontal and vertical scaling [65].

Scaling horizontally means to add more processing nodes to existing systems. For example, scaling out from one web server to multiple web servers that run in parallel and that can handle more requests. These web servers are usually backed by a load balancer system that equally distributes the load between them. Scaling vertically means to add more resources to a single node in a system. For example, increasing the number of CPUs, adding faster CPUs or adding more memory to a single web server. By increasing the processing power of a single node it can process the requests faster and handle more requests in parallel. Figure 4.1 shows the difference between vertical and horizontal scaling techniques.

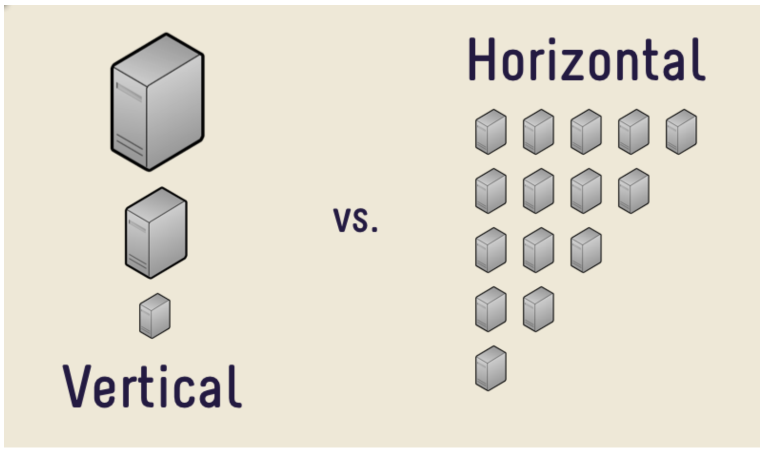


Figure 4.1: Visualization of vertical and horizontal scaling approach.

Scaling horizontally is considered more important as commodity hardware is cheaper compared to the cost of special configuration hardware that powers supercomputers. So adding more nodes into a system is usually more preferable in terms of cost-benefit analysis than vertically scaling and building supercomputers. But at the same time, horizontal scaling increases the complexity of the entire system. To handle additional nodes, the network layer has to be modified, additional load balancers have to be configured, distributed databases have to be set up, etc.

Besides horizontal and vertical scaling, increasing the number of requests that a single node can handle is also an important factor that can increase the scalability without adding modes or buying faster CPUs. By tuning the performance of the server solution in terms of CPU and memory allocation a single node can run more processes in parallel and therefore process more requests. Furthermore, by reducing the resources allocation of the solution, more instances can be run on the same node which can greatly contribute to the scalability factor of the web application and therefore can be also considered a scaling technique. This thesis is researching the latter statement and proposing a CPU and memory optimized server solution that shows improved scalability results.

4.2 High Availability

In computer science, availability represents the probability that a system is operational at a given time, i.e. the amount of time a system is actually operating in relation to the total time it should be operating. Availability can be measured as the ratio between mean time between failures (MTBF) and maximum time to repair or resolve a particular problem (MTTR). The equation (4.1) is used to calculate the availability factor.

$$A = \frac{MTBF}{MTBF + MTTR} \tag{4.1}$$

For example, a system that was unavailable for 1.83 days in a year would then have an availability of 99.5 percent (4.2).

$$A = \frac{365 * 24}{(365 * 24) + (1,83 * 24)} \tag{4.2}$$

$$A = 0,995011313 \tag{4.3}$$

These availability percentages are sometimes referred to by the number of nines or class of nines in the digits. For example, 99,999% of the time would have five nines reliability or class five reliability. More examples are shown in the table 4.1. Availability is most often communicated in documents called service-level agreements or SLA. SLA documents define official commitments agreed between a service provider and a customer which usually include terms about quality, responsibilities and minimal level of availability [66].

Uptime and availability terms are often used synonymously but should not be because they are not the same. A system can be up, but its service

Table 4.1: Examples of class of nines availability

Availability %	Class of nines	Downtime per year	Downtime per month
90%	One nine	36,5 days	72 hours
99%	Two nines	3,65 days	7,2 hours
99,9%	Three nines	8,76 hours	43,8 minutes
99,99%	Four nines	52,56 minutes	4,38 minutes
99,999%	Five nines	5,26 minutes	25,9 seconds
99,9999%	Six nines	31,5 seconds	2,59 seconds
99,99999%	Seven nines	3,15 seconds	262,97 milliseconds
99,999999%	Eight nines	315,569 milliseconds	26,297 milliseconds
99,999999%	Nine nines	31,5569 milliseconds	2,6297 milliseconds

can still be unavailable and unreachable by the user because of a network outage for example. On the other hand, downtime and unavailability terms can be used interchangeably because a system that is in downtime is non-operational and therefore unavailable and unreachable by the user. A system downtime can be either a scheduled downtime usually for the purpose of system maintenance or unplanned downtime which is usually a result of system failure.

High availability systems are those systems that have really high availability rates usually reported in terms of only minutes of downtime per year. Availability features allow the system to stay operational even when errors occur in the system. A highly available system would disable the malfunctioning part and continue operation at a reduced capacity but still available to the users. In contracts, a less capable system might crash and become totally non-operational and unavailable to the users. One principle of reaching high availability is eliminating single points of failure by adding redundancy to the system. This way when a system component fails, the redundant component can take over the work and prevent the failure of the entire sys-

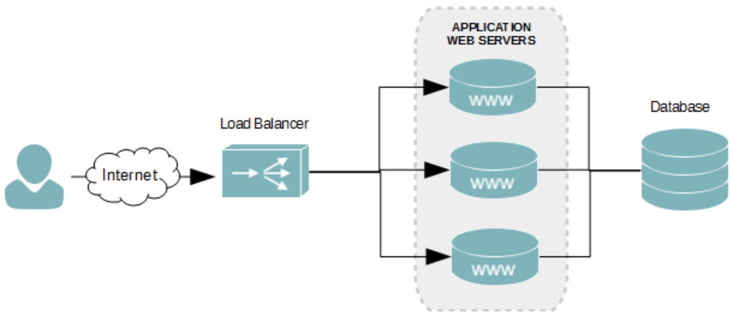


Figure 4.2: Load balancer forwards the requests to web servers.

tem. In computing, this switching to a redundant system upon the failure is called failover. For web applications, this redundancy is usually achieved by using a load balancer system. Multiple redundant servers are connected to a load balancer which has a dual-purpose. First, it balances the traffic equally between the servers and secondly it also takes care of failover in events of a particular server failure. Using load balancers instead of single servers increases reliability and availability of web applications [67]. Figure 4.2 shows how a load balancer handles incoming traffic and forwards the requests to available web servers.

Another important concept in highly available systems is fault tolerance that describes the ability of a system to continue operating properly in the event of one or multiple faults within the system. This ability to maintain functionality when parts of a system breakdown is also referred to as graceful degradation. Fault tolerance can be achieved by anticipating exceptional conditions and building the system to cope with these conditions. This means building software solutions that are robust and designed to continue operating despite an error, exception, or invalid input happens. In events of these exceptional conditions happening, the solution has to cope with them and converge towards an error-free state which is also called self-stabilization [68].

Based on the formula for high availability mentioned before, there are two ways to improve availability. Either by extending the mean time before fail-

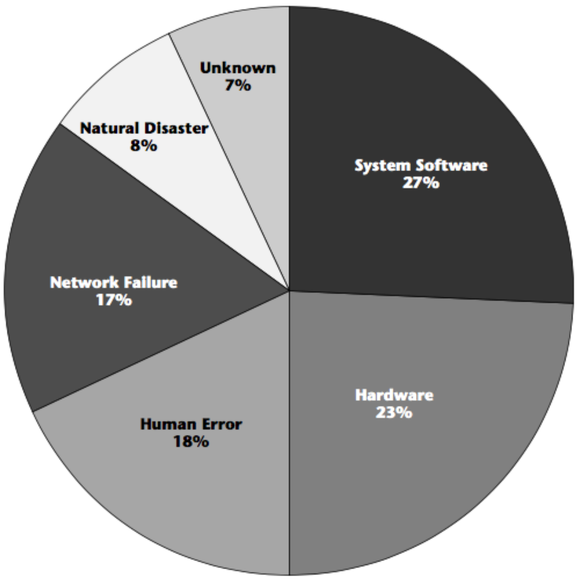


Figure 4.3: The most common causes of unplanned downtime [69].

ure or by extending the mean time to recover. By improving either of these measures the availability factor of the system improves. To improve MTBF the system has to become more fault tolerant to prevent complete system failure and to gracefully degrade to an error-free state. Building redundant systems and composing load balancers implies improving the hardware capabilities of the whole system to be more tolerant to failure and therefore improves MTBF. Figure 4.3 shows the most common downtime causes and software failure accounts for 27% of the causes. This means that building fault-tolerant software is as much important as having load balancers and redundant servers [69].

This thesis is researching the latter statement and proposing a software solution that improves MTBF by optimizing scalability and thus increasing the time before the system fails.

4.3 Impact on real-time web applications

Both scalability and high availability are very important for real-time applications. As noted before, the bidirectional communication nature of real-time web applications results with much higher loads to the server solutions compared to regular nonreal-time web applications. These server solutions are often pushed to their limits while expected to run without failures. Therefore, these server solutions that are handling all the requests and responses have to be scalable to be able to cope with high traffic and also fault tolerant to gracefully handle exceptional conditions and keep the service always available and operational.

Scalability and high availability can be achieved using different techniques like introducing redundancy and failover units with load balancers or clusters. These redundancy units give the server more processing power to handle more requests and bring failover mechanisms that improve fault tolerance. But this thesis researches and alternative approach to achieving scalability and high availability. This alternative approach proposes using software methods instead of hardware solutions to improve scalability and high availability.

A variety of programming languages can be used for developing software solutions but not all are equally suited for real-time applications. Most of these languages provide some basic scalability and high availability features but some are better than others. One programming language called Erlang is known especially for its high availability, fault tolerance, and distributed scalability.

4.4 Erlang

Erlang is a functional programming language developed in Ericsson company by Joe Armstrong, Robert Virding and Mike Williams in 1986. It was originally a proprietary language owned by Ericsson but was later released as open source in 1998. Erlang was designed to support massively scalable soft real-time systems with requirements on high availability. It is actively used in telecoms, banking, e-commerce, computer telephony and instant messaging applications. Erlang's runtime system has a built-in support for concurrency, distribution and fault tolerance. In 1998 Ericsson announced the AXD301 switch written in Erlang and reported to achieve a high availability of nine nines (99.9999999%) which equals only 31ms of downtime a year while handling 30-40 million calls per week [70].

In his doctoral dissertation called "Making reliable distributed systems in the presence of software errors", Joe Armstrong addresses the problem of constructing a reliable system from programs which may themselves contain errors. To make a reliable system from faulty components places certain requirements on the system. The requirements can be satisfied, either in the programming language which is used to solve the problem or in the standard libraries which are called by the application programs to solve the problem. He describes the Erlang programming language and other library modules written in Erlang that satisfies the required characteristics and presents solutions for fault-tolerant programs [71].

Erlang's main strength is support for concurrency and the powerful set of primitives for creating processes and managing communication among them. Erlang belongs to the family of pure message passing languages which means it is a concurrent process-based language having strong isolation between concurrent processes. These Erlang processes are neither operating system processes nor operating system threads, but lightweight processes that are created and managed by the Erlang's virtual machine (VM) called BEAM. These processes are quite different from threads which have to share resources between themselves and therefore it is extremely difficult to isolate components from each other. Without isolation, errors in one component can propagate to another component and damage the internal consistency of the system. By using message passing communication between processes, Erlang is better in isolating components and therefore less error-prone. Furthermore, Erlang provides language-level features for creating and managing these processes which simplify concurrent programming and remove the dependency for an external library support like threads. Also, as threads communicate using shared variables explicit locking schemes are required to prevent deadlocks and unpredicted behavior when multiple threads are simultaneously reading and writing into shared variables. Erlang removes this requirement on locking by using inter-process communication via a shared-nothing asynchronous message passing. Removing the locking requirement also means improving the language capabilities for concurrency and scalability making Erlang excel at that.

Considering all mentioned capabilities of the Erlang programming language, it can be considered a valid choice for developing server solutions that have to manage real-time web applications. Back when Erlang was created in 1986, telephony was one of the biggest global systems that had to be scalable, fault-tolerant and highly available to support millions of calls happening simultaneously global wide. Erlang was designed to support exactly that kind of systems. Today, real-time web applications share the same problem domain. They have to be scalable, fault-tolerant and highly available to handle enormous loads created by millions of users using the application simultaneously. Some of the most interesting cases of Erlang usage in real-time applications are Facebook Messenger and WhatsApp which are probably the biggest real-time chat applications in the world. Facebook Messenger application with 800 million users and WhatsApp with 1 billion users both reported that their Erlang powered stack processes more than 50 billion messages per day with a greater than 99.9% availability [72]. Erlang and its ecosystem called Open Telecom Platform (OTP) provide a collection of useful middleware, libraries, and tools that make it easier to create programs for the telecom domain but was never designed with web application server solutions in mind. This makes developing web applications in Erlang difficult because it is missing a lot of components that other modern web-related languages provide like Rails, Django, Spring, etc. [73] [74] [75].

In 2011, Jose Valim created a new programming language called Elixir which is a functional, concurrent, general-purpose language that runs on the

Erlang VM BEAM. Elixir has a built-in compiler that compiles Elixir code directly to Erlang bytecode that can be run on the Erlang VM. This makes Elixir share the same abstraction for building distributed, fault-tolerant applications as in Erlang but using a more understandable, modern and fresh syntax. By running everything inside lightweight processes that are isolated in a similar way as Erlang, hundreds of thousands of processes can be run concurrently on the same machine making the solutions easily vertically scalable. Also, as all the communication is message based and no shared variables are used, processes can even communicate when being run on different machines on the same network providing the foundation for distribution and great horizontal scaling possibilities. Furthermore, fault tolerance is also provided by default where mechanisms inherited from Erlang-like supervisors allow restarting parts of the system that go awry and revert the system state to a stable condition that is guaranteed to work [76].

Elixir aims to modernize and improve the experience of developing Erlang-powered systems. The language is a compilation of features from various other languages such as Erlang, Clojure, and Ruby. Elixir ships with a toolset that simplifies project management, testing, packaging, and document building which altogether lower the entry barrier into the Erlang world and improve developer productivity. At the same time having the Erlang runtime as the target platform means Elixir-based systems are able to use all the libraries from the Erlang ecosystem, including all the battle-tested tools for concurrency and fault tolerance that ship with Erlang. To facilitate web application development special Elixir based frameworks have been created where the most famous one is probably the Phoenix web framework developed by Chris McCord in 2015. Phoenix framework provides tools and best practices for building modern web applications in Elixir. It is influenced by similar popular web frameworks, like Rails for the Ruby programming language, and allows creating fast, concurrent and reliable web applications that are actually running on the Erlang VM [77].

Chapter 5

Client-side prototypes

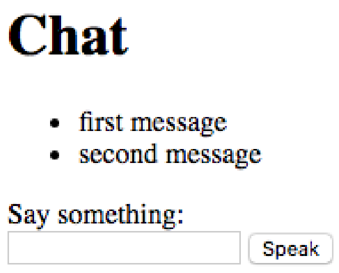
5.1 Architecture

Four simple web application solutions will be presented that are using polling, long polling, streaming, and WebSocket for real-time web behaviors. These applications will be used to compare the advantages and disadvantages of mentioned client-side approaches while handling real-time communication. All the solutions will have the same architecture - a simple chat application that enables multiple users to exchange messages in real-time.

The chat application has a chat room which users can join, read existing messages and create new messages. The interface of the chat application is quite simplified and has a title, list of existing messages and a form for submitting new messages as shown in the figure 5.1.

After the user submits the new message, by clicking the `Speak` button, the message appears on the list as a new message. The same message should instantly appear in any other user's chat application being opened at that moment. This real-time behavior is what the prototype application is designed to present.

Because the purpose of these prototypes is only to observe data exchange and client-server request cycles, some chat related features have been omitted. No authorization and authentication mechanisms have been implemented


Chat

- first message
- second message

Say something:

Figure 5.1: Interface of the chat application.

therefore all users can enter the chat room without any login step or credential checks. Also, all users share the same chat room and create anonymous messages. Furthermore, no additional visual styling has been applied to the HTML elements in the chat room except the default browser element styling. When building a real chat application, these features should not be omitted and should be implemented in the final product but as this is only a prototype application focused on showing performance implications of different real-time techniques these features have been omitted.

5.2 Polling prototype

The polling prototype application presents the idea of having regular polling interval that sends requests to the server to check for changes. The application has three endpoints:

- /chat
- /chat/speak

- /chat/poll

The `/chat` endpoint is where the chat room can be accessed showing the list of existing messages and the form for creating new messages. The form submits new messages to the `/chat/speak` endpoint which creates these messages in the database and the `/chat/poll` endpoint is used for interval-based polling check for changes. All the interactions on the client are controlled using the JavaScript programming language.

```javascript
function poll() {
  var after = $('#messages li:last-child').attr('data-id');

  $.get('/chat/poll', { after: after }, function(data) {
    $.each(data, function(_, value) {
      $('#messages').append("<li data-id='" + value['id'] + "
          '>" + value['message'] + "</li>");
    });
  })
  setTimeout(poll, 1000);
}
```

The `poll()` function makes requests to the server every one second to check for new messages. In order to ask for new messages, the client first has to figure out what is the last message it already has. That way it can ask the server to return all the messages that have been created after that exact message. This is really important in the polling scenario because multiple messages can be created between two consecutive polling requests and this is the best way to communicate what messages the client already has. The message ID is a unique identification of every message and a perfect candidate to be used for this purpose. The function then makes a GET request to the `/chat/poll` endpoint sending together the ID of the last message as the query parameter. The server will return all the messages with the higher ID which represents all the messages that have been created in between the last poll request and the current one. The returned messages are inserted into the chat room with their IDs so that the next poll request can send the newest message ID available.

```
$(document).on('submit', '[data-behavior~=chat_form]',
    function(event) {
  $input = $('[data-behavior~=chat_message]');

  $.post('/chat/speak', { message: $input.val() }, function(
      data) {});

  $input.val('');
  event.preventDefault();
});
```

When the `speak` button on the form is clicked, the form gets submitted and the submit event gets called. The value from the input field in the form gets collected and posted to the `/chat/speak` endpoint using the `.post()` method. The input gets cleared and the form is reset and ready for the next interaction. The server gets the POST request on the `/chat/speak` endpoint and creates a new message in the database. On the next poll request, the client will receive the message it previously created and it will be inserted into the chat room.

5.3 Long polling prototype

The long polling prototype application presents the idea of using HTTP long polling mechanism that holds the response until the changes are available. The application has three endpoints:

- /chat
- /chat/speak
- /chat/poll

The `/chat` endpoint is where the chat room can be accessed showing the list of existing messages and the form for creating new messages. The form submits new messages to the `/chat/speak` endpoint which creates these messages in the database and the `/chat/poll` endpoint is used for long polling check for changes. The routing structure of the application is identical to the previous polling example. All the interactions on the client are controlled using the JavaScript programming language.

```
function poll() {
  $.get('/chat/poll', function(data) {
    $('#messages').append("<li>" + data.message + "</li>");
    setTimeout(poll, 1);
  })
}
```

The `poll()` function makes a GET request to the `/chat/poll` endpoint on the server and waits for the response. The server does not respond immediately but keeps the request pending until it has any data to send, i.e. until a new message is created. When the client ultimately receives the data, the included message is inserted into the chat room and immediately a new GET request to the `/chat/poll` is sent. This ensures that the client always has an open outstanding request with the server waiting for new messages. Because long polling is not using an interval for polling changes but opens a new connection right away, it does not have to worry about messages being created in between the polling requests. Therefore, it does not have to calculate the last message ID and can skip sending the query parameter with the requests.

```
$(document).on('submit', '[data-behavior~=chat_form]',
    function(event) {
  $input = $('[data-behavior~=chat_message]');

  $.post('/chat/speak', { message: $input.val() }, function(
      data) {});

  $input.val('');
  event.preventDefault();
});
```

The form handling behavior is very similar to the polling example. The `speak` button submits the form, the value of the input field is collected and posted to the `/chat/speak` endpoint and the form gets cleared. As soon as the server receives the POST request and creates the new message, the outstanding request which was already on hold gets fulfilled and returned to the client with the data about the new message. This is significantly different compared to the previous polling example where the response is received only after the next polling request. Immediately after receiving the response, the client sent the next request to the server thus closing the long polling cycle.

5.4 Streaming prototype

The streaming prototype application presents the idea of using HTTP streaming mechanism which uses a single request that is kept open indefinitely and the data is streamed over that same request. The application has three endpoints:

- /chat
- /chat/speak
- /chat/stream

The `/chat` endpoint is where the chat room can be accessed showing the list of existing messages and the form for creating new messages. The form submits new messages to the `/chat/speak` endpoint which creates these messages in the database and the `/chat/stream` endpoint is used for opening the long-living request that streams the changes to the client. All the interactions on the client are controlled using the JavaScript programming language.

```javascript
function stream() {
  var eventSource = new EventSource("/chat/stream");

  eventSource.addEventListener('refresh', function(event) {
    $('#messages').append("<li>" + event.data + "</li>");
  });
}
```

The `stream()` function opens a new EventSource stream to the `/chat/stream` endpoint on the server. As in the previous long polling example, there is no need for calculating and sending last message ID parameter with the request because a single request is open indefinitely and no polling interval is used. When a comment is created on the server it is immediately sent to the client using the opened stream. The client listens on the stream and waits for the refresh event to happen and then inserts the received message data into the chat room. Compared to the long polling example, after the data is received the request is not closed and the stream is kept alive. This ensured that the client always has an open stream with the server waiting for new messages.

```
$(document).on('submit', '[data-behavior~=chat_form]',
    function(event) {
  $input = $('[data-behavior~=chat_message]');

  $.post('/chat/speak', { message: $input.val() }, function(
      data) {});

  $input.val('');
  event.preventDefault();
});
```

The form handling behavior is very similar as in both previous examples. The `speak` button submits the form, the value of the input field is collected and posted to the `/chat/speak` endpoint. When the server receives the POST request it creates the new message in the database and immediately sends data containing the new message back to the client, using the already available stream. Similarly, as in the long polling example, the response is immediate because it does not depend on the polling interval. One additional advantage here is that the request is never closed so there is no need for sending consecutive requests which removes the overhead of opening new connections.

5.5 WebSocket prototype

The WebSocket prototype application presents the idea of using WebSocket mechanism for exchanging the data between the server and the client over an independent TCP socket connection. The application has three endpoints:

- /chat
- /chat/speak
- /chat/socket

Same as in the previous examples the chat is accessed on the `/chat` endpoint and form is submitted to the `/chat/speak` endpoint which creates these messages in the database. The `/chat/socket` endpoint is used to establish the WebSocket connection and to facilitate data exchange between the server and the client. All the interactions on the client are controlled using the JavaScript programming language.

```
function socket() {
  var webSocket = new WebSocket("ws://localhost:3000/chat/
     socket");

  webSocket.onmessage = function (event) {
    $('#messages').append("<li>" + event.data + "</li>");
  }
}
```

The `socket()` function open a WebSocket connection to the `/chat/socket` endpoint on the server. The HTTP request is upgraded and switched to the WebSocket protocol which is now opened between the server and the client. When a comment is created on the server it is immediately sent to the client using the opened socket. The client listens on the socket changes using the `onmessage` event handler. When a message is received the event handler is triggered and messages are inserted into the chat room. As in the previous streaming example, a single request is used for all data exchanges, but the difference is that WebSocket example is not using HTTP for data transfer but a more lightweight WebSocket protocol. As in the both previous examples, there is no need for calculating and sending last message ID parameter with the request because no polling interval is used and all the changes are received immediately as they happen.

```
$(document).on('submit', '[data-behavior~=chat_form]',
   function(event) {
  $input = $('[data-behavior~=chat_message]');

  $.post('/chat/speak', { message: $input.val() }, function(
     data) {});

  $input.val('');
  event.preventDefault();
});
```

The form handling behavior is again quite similar to all the previous examples. The `speak` button submits the form, the values from the input is collected and posted to the `/chat/speak` endpoint which creates new messages in the database. As soon as the messages are created they are also sent back to the clients using the opened socket connection.

Chapter 6

Server-side prototypes

6.1 Architecture

Two simple server prototypes will be presented with the same architecture - a simple chat application where users can exchange messages in real-time. One prototype solution will be developed using the Rails web framework for the Ruby programming language and the second prototype solution will be developed using the Phoenix web framework for the Elixir programming language.

The client part of the application will be using the WebSocket approach with an interface quite similar to previous prototypes. The chat application shows a list of existing messages and a form for submitting new messages. The messages that user submits are immediately broadcasted to all the other users connected to the chat application at that moment.

As in the previous prototypes, many chat related features like authorization, authentication, visual styling, etc., have been omitted for brevity reasons.

6.2 Ruby on Rails prototype

Ruby on Rails, or simply Rails, is a server-side web application framework written in Ruby that uses well-known software engineering patterns and paradigms like convention over configuration (CoC), and don't repeat yourself (DRY) for building ambitious server solutions. The Rails framework provides a handful of tools and libraries which are used as building blocks for developing web applications. By using these building block, developing web applications like this chat application becomes much faster, easier and less error-prone.

This prototype application is intended to present the real-time capabilities of applications written in Rails framework and most of the other functionalities of the framework will be omitted for brevity.

The prototype has two endpoints:

- /chat
- /cable

The `/chat` endpoint is where the chat room can be accessed showing the list of existing messages and the form for creating new messages. The `/cable` endpoint is used to establish the WebSocket connection that facilitates the data exchange between the server and all the connected clients. In comparison to the WebSocket prototype application from chapter 5.5, the `/chat/speak` endpoint is not required because the form submits new messages using the existing WebSocket connection to the `/cable` endpoint. As WebSocket communication is full duplex it is not limited only to server-to-client communication. The client can also send requests to the server using the same WebSocket connection. This way the application needs one less endpoint for handling HTTP requests and uses the faster WebSocket connection to submit new messages. All the interactions on the client are controlled using the JavaScript programming language.

```
$(document).on('submit', '[data-behavior~=chat_form]',
    function(event) {
```

```
  $input = $('[data-behavior~=chat_message]');

  App.chat.speak($input.val());

  $input.val('');
  event.preventDefault();
});
```

The `speak` button submits the form, the value of the input field is collected, sent to the server, and the input gets cleared. The message is sent to the sever using the existing WebSocket connection by calling the `App.chat.speak()` abstraction which will later be explained in more detail.

When the user first opens the application it has to establish the WebSocket connection to the server. The WebSocket connection is opened automatically by the Rails framework. The framework also automatically handles reconnections if any interruptions with the connection happen. This way the programmers do not need write additional code to cope with those situations because the framework will handle that. Also, the Rails framework provides a lot of helpful abstractions for handling WebSocket communication in both client side and server side code. The `cable` is an abstraction for a single WebSocket connection. Each `cable` can have multiple `channels` which represent a single logical topic where consumers can subscribe and send or receive messages.

This prototype application has a single channel called `ChatChannel` where all the users connect to send and receive chat messages.

```
class ChatChannel < ApplicationCable::Channel
  def subscribed
    stream_from 'chat_channel'
  end

  def speak(data)
    @message = Message.create(text: data['message'])

    ActionCable.server.broadcast 'chat_channel', message:
        @message.text
  end
end
```

This `ChatChannel` definition has two actions: `subscribed()` and `speak()`. The `subscribed()` action is triggered when a consumer first connects to the channel and it ensures that all the users get subscribed to this channel to receive updates from the channel when an action is triggered. The `speak()` action is triggered when the message is sent from a client to the server using the WebSocket connection. It receives the message, persists that message to the database and broadcasts the same message to all the existing consumers connected to the channel. This triggers WebSocket requests for all the connected users and they all receive the newly created message as payload.

```
App.chat = App.cable.subscriptions.create("ChatChannel", {
  received: function(data) {
    $('#messages').append("<li>" + data['message'] + "</li>")
      ;
  },

  speak: function(message) {
    return this.perform('speak', { message: message });
  }
});
```

When the message is received by the client, the `received()` method is called and the content of the message gets inserted into the list of chat messages. The `speak()` method, already mentioned before, is called when the user submits the form and it is an abstraction that actually sends the request to the server over the existing WebSocket connection. The request has a structured format that indicates used channels, triggered actions, and holds the data. When the user inputs ”this is a message” and submits the form the following request is generated:

```
{
  "command": "message",
  "identifier": {
    "channel": "ChatChannel"
  },
  "data": {
    "message": "this is a message",
    "action": "speak"
  }
}
```

The channel name and the action name that needs to be triggered are indicated, and the message payload is included. When the server receives this request it calls the `ChatChannel` class mentioned before, and triggers the `speak()` action with the provided message as data. The same action then broadcasts the newly created message to all the channel subscribers. This broadcast is implemented as a WebSocket event sent to all the open connections and it also has a structured format that indicates the channel name and holds the data.

```
{
  "identifier": {
    "channel": "ChatChannel"
  },
  "message": {
    "message": "this is a message"
  }
}
```

When the client receives this event it can easily infer to what channel this message belongs by using the identifier field. In the above case, it calls the `ChatChannel` and triggers the `received()` method, already mentioned before, and passes the message body ”this is a message” that then gets inserted into the list of chat messages.

6.3 Phoenix prototype

Phoenix is a server-side web application framework written in Elixir running on the Erlang VM. Similarly as Elixir was influenced by Ruby, Phoenix was greatly influenced by Rails which means they share a lot of well-known software engineering patterns and paradigms that facilitate developing web applications. Phoenix combines all the best practices for web development with speed and robustness of Erlang making it one of the most ambitious open source web application frameworks today. Same as in the previous Rails application, only real-time capabilities of the application will be showcased and most of the other functionalities of the Phoenix framework will be omitted for brevity.

The prototype application has two endpoints:

- /chat
- /socket

The `/chat` endpoint is the same as in the previous example, shows the chat room, lists the existing messages and present the form for creating new messages. The `/socket` endpoint is used for WebSocket connections and full-duplex message exchange between the server and the clients. Same as in the previous example, the `/chat/speak` endpoint is not required because the form submits new messages using the existing WebSocket connection and not normal HTTP requests. Both prototype applications are designed as similarly as possible to make the tests and measurements valid and comparable. All the interactions on the client are controller using the JavaScript programming language.

```
var channel = Chat.init(socket);

$(document).on('submit', '[data-behavior~=chat_form]',
   function(event) {
  var $input = $('[data-behavior~=chat_message]');

  channel.push("speak", {body: $input.val()})

  $input.val('')
  event.preventDefault();
});
```

The `speak` button submits the form, the value of the input field is collected, sent to the server, and the input gets cleared. The message is sent to the server using the existing WebSocket connections by calling the `channel.push()` function and sending the `speak` event and the message as the body of the request.

When the application is first opened it has to establish the WebSocket connection to the server. In the previous prototype, the Rails framework handled this automatically and no additional code had to be implemented by the programmer. The Phoenix framework is less invasive and does not magically invoke WebSocket connections but provides libraries built on best practices which a programmer can use to facilitate these actions. Similarly as in the Rails framework, WebSocket connections can be organized into multiple channels where each represents a single topic where consumers send and receive messages.

This prototype application also has a single channel called `ChatChannel` where all the users connect to send and receive chat messages.

```elixir
defmodule PhoenixChat.ChatChannel do
  use PhoenixChat.Web, :channel

  def join("chat", _payload, socket) do
    {:ok, socket}
  end

  def terminate(_reason, _socket) do
    :ok
  end

  def handle_in("speak", payload, socket) do
    {_status, message} = %PhoenixChat.Message{text: payload["body"]} |> Repo.insert

    broadcast! socket, "publish", %{body: message.text}
    {:noreply, socket}
  end
```

```
end
```

This `ChatChannel` definition has three actions: `join()`, `terminate()` and `handle_in()`. The `join()` action is triggered when a consumer first connection to the channel and `terminate()` is called when the consumer disconnects from the channel. In this implementation, both actions just return a success state and handle no other responsibilities. The `handle_in()` action is triggered when the WebSocket event of type speak is received. It then persists the received message to the database and broadcasts the same message to all the existing consumers on the channel as the publish event. All the connected consumers receive a WebSocket request with the publish event and the message body included in the payload.

The following JavaScript implementation handles joining the channel and receiving messages:

```
class Chat {
  static init(socket){
    var channel = socket.channel("chat", {})

    channel.join();

    channel.on("publish", data => {
      $("#messages").append("<li>" + data['body'] + "</li>");
    })

    return channel;
  }
}
export default Chat
```

It opens and joins a channel called `chat` on the existing socket connection and registers a handler for receiving messages. When the WebSocket message having an event of type `publish` is received, the handler is triggered and the message body gets appended to the list of messages in the chat. As in the previous Rails prototype application, both the request and the response have a structured format that indicates used channels, triggered actions, and hold the data. When the user inputs ”this is a message” and submits the form the following request is generated:

```
{
  "topic":"chat",
  "event":"speak",
  "payload":{
    "body":"this is a message"
  }
}
```

The structured format is even simpler than in the Rails prototype application. It indicates the topic channel name (chat), the event to be triggered (speak) and has the message body in the payload. When the server receives this request it calls the `ChatChannel` class and triggers the `handle_in()` action providing the `speak` as the event type and the message body as the payload. The same action then broadcasts the newly created message to all the channel subscribers. This broadcast is again a WebSocket event with the type of `publish` sent to all open connections and its format is structured as follows:

```
{
  "topic":"chat",
  "event":"publish",
  "payload":{
    "body":"this is a message"
  }
}
```

It indicates the topic to be the `chat` channel, the event to be `publish` and carries the body of the message in the payload. When the client receives this event it triggers the `.on('publish')` event handler that was described before and the message ”this is a message” gets inserted into the list of chat messages.

Chapter 7

Testing client prototypes

7.1 Methodology

All the client prototypes will be tested using the same test scenario where an automated script will be used to simulate real user interaction. The chat application will be opened in two browsers and an automated script will simulate interaction by entering predefined text into the input filed and submitting the form.

```
function automate() {
  $('[data-behavior~=chat_message]').val('this is a chat
     message!');
  $('[data-behavior~=chat_form] :submit').click();
  setTimeout(automate, 2500);
}
```

The `automate()` function enters the text `this is a chat message!` into the input field and then submits the form by clicking on the form's submit button. This simulates the interaction between the user and the chat application similarly as a real user would behave in the browser. The function is repeated in intervals of 2500 milliseconds to produce repeated behavior and simulate traffic that can then be measured. A fixed interval and the same predefined message are used in every interaction to ensure that the measurements can be compared between all prototype applications.

At the same time, network traffic will be sniffed and analyzed using a tool called Wireshark [78] [79]. Parameters like the number of packets, average packets per second, average package size, the total size of traffic in bytes, and average bytes per second will be compared to determine the differences between prototype applications and their impact on traffic and latency. All tests will be running for 60 seconds which is a long enough interval to capture sufficient data that can then be analyzed. Given two parallel opened applications each submitting the form every 2500 milliseconds gives a total of 48 messages that get created during each test.

7.2 Results

The table 7.1 shows summarized data collected by running the test on all four prototypes.

From the data, it can be assumed that every presented solution takes precedence over the previous one in terms of performance gains.

Polling is the simplest solution that works on every browser but has the worst performance. Since the foundation of polling applications is interval based polling it is expected that this prototype will generate the highest amount of traffic data.

Long polling shows improvements compared to polling where the total number of packets is reduced from 918 to 555. This is primarily because the long polling solution eliminated the overhead of interval based requests that poll for changes and often result in empty results. The total number of bytes transferred is reduced by around 20% which is also due to the removal of vain polling requests. Furthermore, the density of data is increased from 329,2 bytes to 442,2 bytes per package because the smaller number of packages basically transferred the same amount of messages created during the test. In general, the long polling prototype solution shows better results and improved performance compared to the polling prototype solution.

Streaming proved better than both polling and long polling in terms

Table 7.1: Collected results from testing client prototypes

	Packets	Bytes	Bytes/s	Bytes/package
Pooling	918	302211	5037	329,2
Long-pooling	555	245338	4088	442,2
Streaming	1396	186046	3100	133,3
WebSocket	568	138492	2300	243,8

of performance where the total number of bytes transferred is reduced by around 25% over long polling prototype and around 40% over the polling prototype. The data reduction is the consequence of removing the overhead of opening new connection after every request and using a single long-living request to stream the data. This results in a more lightweight transfer of the same amount of messages in only 3100 bytes per second which contributes to the improved latency of the solution. The total number of packets is increased due to the way data is packed for streaming but it does not affect the performance of the streaming solution. In general, the streaming prototype solution shows better results and improved performance compared to both previous polling solutions.

WebSocket shows the best performance improvements among all the presented solutions where the total number of bytes transferred is reduced by around 25% over the streaming prototype, around 44% over the long polling prototype and around 55% over the polling prototype. The same amount of message data is transferred by only 2438 bytes per second which is a direct consequence of not using HTTP for data transfer but a more lightweight WebSocket protocol. The total number of packets is significantly decreased compared to the streaming prototype because the data is more efficiently packed. In general, the WebSocket prototype shows better results and improved performance compared to all previous prototypes presented.

Taken into consideration all the mentioned drawbacks of polling, long polling and streaming approach it can be concluded that the WebSocket approach is superior in every category and is currently the best methodology which can be used for developing real-time web applications.

Chapter 8

Testing server prototypes

8.1 Methodology

In this chapter two server prototypes will be put to test by simulating the real-time interaction. For an application to simulate real-time behavior it has to generate a large number of interactions with the server. Whether the requests come as pooling calls from the client, or the server independently streams the data, the server should observe an increased load when serving a real-time application. This total number of requests that a server has to accept and respond to can be defined as a measure called requests per minute (RPM).

For web applications in general, scalability can be examined as a measure of the number of RPM that an application can effectively support without getting overflown. For observed real-time prototype applications, this can be translated to the number of successfully opened WebSocket connections on the server at the same time. For every open WebSocket connection, the server has to allocate some of its resources like CPU and memory to keep the connection open and be ready to respond to requests. As every server has a limited amount of resources, at one point it cannot accept any new WebSocket connections due to resource exhaustion and starts to decline connections. This total number of successful WebSocket connections to the server is what these tests are going to measure. Also, the test will measure the time before the system starts throwing errors because of resource exhaustion which will later be used to calculate the Mean Time Before Failure (MTBF).

8.2 Distributed load testing

To measure the total number of connections on the server, these connection have to be generated and sent to the server. The most accurate way to simulate load is to actually use thousands of clients that open WebSocket connections towards the server. This accurately represents how real-time applications work where thousands of users open WebSocket connections to the server from their browser or similar devices. But this approach is too complex and unsustainable for methodical testing because the testing environment has to control thousands of client at the same time.

A better approach is to use two testing machines where one behaves as a client and the second behaves as the server. From the server's perspective, these two approaches are identical. The server has to accept these requests, open WebSocket connections, keep them open and ready for receiving messages regardless of who actually made these requests.

When load testing real-time applications, thousands of requests is not a large number of requests and sometimes hundreds of thousands of requests have to be generated to actually stress out the server. In that case, a single client machine is not capable enough to generate this amount of requests so several client machines are used to generate load on a single server. This method of load testing is called distributed load testing [80] and is shown in the figure 8.1.

For performing distributed load testing, different tools can be used like Goad [81], Artillery [82], Locust [83], JMeter [84] and Tsung [85]. Most of these tools support only stress testing web protocols like HTTP but JMeter and Tsung are the most versatile and support a handful of protocols like SOAP, LDAP, FTP, SMTP... For testing the prototype applications, Web-

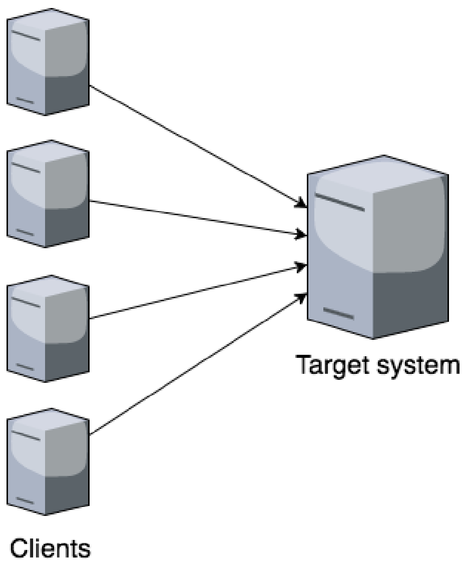


Figure 8.1: Distributed load testing architecture.

Socket connections have to be generated so both JMeter and Tsung can be used. Although they are quite similar in terms of options they offer, Tsung is much more performant and faster because it runs on Erlang whereas JMeter runs on Java, so Tsung will be used.

Tsung is an open source distributed load testing tool that can be used to stress load protocols like HTTP, WebDev, SOAP, PostgreSQL, MySQL, LDAP, Jabber/XMP and few others. The purpose of Tsung is to simulate requests in order to test the scalability and performance of IP based client/server applications. It can be distributed on several client machines and is able to simulate hundreds of thousands of virtual users concurrently. With enough hardware, even millions of concurrent connections can be simulated [86].

The client instance will run a series of load tests by generating a fixed number of requests per second for 300 seconds. Each test will increase the number of requests per second until the server instance hits the resource limit and stops accepting new requests. For the duration of each test, measurement like network statistics, response times, throughput, and total successful connection count will be taken to be analysed and compared. These tests scenarios are configured using the Tsung XML configuration files which will be explained alongside each test.

8.3 Cloud deployment

The testing environment for distributed load testing can be set up either using local network and physical machines connected to that network or using a cloud-based service where everything is set up inside a cloud. The cloud option is much more flexible and scalable because different scenarios can easily be simulated just by changing few variables and spawning new server instances. Compared to using physical hardware for simulating load testing, various combinations of hardware have to be bought to perform different scenarios, which can be very expensive. One of the biggest cloud service providers today is Amazon Web Services (AWS) which offers different services and options like servers, storage, databases, load balancers, DNS, CDN, etc.

For this distributed load testing environment, two instances of Elastic Compute Cloud (EC2) will be used. One instance will be used as a client that generates requests and the other instance will be used as a server that accepts and handles the requests. An important thing here is to set up these instances in the same AWS zone so that the latency factor for communication between instances is minimal. Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. They have varying combinations of CPU, memory, storage, and networking capacity, which gives the flexibility to choose the appropriate mix of resources which best suits the application needs (shown in table 8.1). This is also great for testing where different scenarios can be easily set up, tested and compared. For this testing purposes, two t2.micro instances will be created each having 1 CPU (2.5 GHz Intel Xeon Family) and 1GB of memory.

After the instances are up and running on AWS, software packages have

Table 8.1: AWS EC2 instance types

Instance	CPU	Memory (GB)
t2.nano	1	0.5
t2.micro	1	1
t2.small	1	2
t2.medium	2	4
t2.large	2	8
t2.xlarge	4	16
t2.2xlarge	8	32

to be installed in order to run the prototype applications and the Tsung client on the testing machines. The installation details on both instances will be only superficially explained without going into details for the sake of brevity. On the client instance Erlang, Tsung and several other software packages required for compiling and running Erlang have to be installed. The server instance needs two independent setups. One for running the Rails prototype application where Ruby environment needs to be installed, and one for running the Phoenix prototype application where Erlang and Elixir environments need to be installed.

After the setup is complete the server solutions have to be deployed to the server. There exists a variety of tools that mitigate deploying Rails and Phoenix applications to cloud services like Capistrano [87] [88], continuous integration and deployment platforms like Semaphore [89] and Codeship [90] or even cloud application platforms like Heroku [91] which make deploying applications effortlessly.

For testing purposes in this thesis, no additional tools will be used and the solutions will be simply git pulled to the server and run manually.

When the server instances are set up properly and the prototype applications are deployed to the server, they can be run and should be available at the URL address:

- http://server-ip:3000/chat - Rails prototype
- http://server-ip:4000/chat - Phoenix prototype

8.4 Ruby on Rails prototype tests

The load testing scenario is based on the Tsung XML configuration file located on the client instance. The following XML file is used for testing the Rails prototype application:

```xml
<!DOCTYPE tsung SYSTEM "/usr/local/Cellar/tsung/1.6.0/share/
   tsung/tsung-1.0.dtd">
<tsung loglevel="debug" version="1.0">
  <clients>
    <client host="localhost" use_controller_vm="true"
       maxusers="30000" />
  </clients>

  <servers>
    <server host="35.162.132.55" port="3000" type="tcp" />
  </servers>

  <load>
    <arrivalphase phase="1" duration="300" unit="second">
      <users maxnumber="30000" arrivalrate="10" unit="second"
          />
    </arrivalphase>
  </load>

  <options>
    <option name="ports_range" min="1025" max="65535"/>
  </options>

  <sessions>
    <session name="websocket" probability="100" type="
       ts_websocket">
      <request>
        <websocket type="connect" path="/cable"></websocket>
      </request>

      <request>
        <websocket type="message" frame="text">
          {"command": "subscribe", "identifier": "{\"channel\
             ":\"ChatChannel\"}"}
        </websocket>
      </request>

      <for var="i" from="1" to="100" incr="1">
        <thinktime value="25"/>
      </for>
    </session>
  </sessions>
</tsung>
```

First, the client and server instance locations are configured. Tsung supports running simultaneous tests using multiple client instances which are required when generating hundreds of thousands of requests but for this load testing one client is sufficient so it is set to be the same instance - `localhost`. The server is located on the IP address 35.162.132.55 and the port 3000 is specified because on that port the Rails application is running. Next, the actual load scenario is specified. Only one load phase is used where for the duration of 300 seconds 10 new requests are generated each second. Which should give a total number of 3000 requests when the load test is finished. This way the server instance is stressed gradually adding 10 more requests every second which is ideal for taking measurements and observing server load.

In the sessions part, the behavior of each request is defined. Each request will first try to connect to the WebSocket interface on the `/cable` path and after is succeeds one message will be sent over the connection. The format of the messages is structured in the way that it requests to join the channel called `ChatChannel` on the server. After sending the message, the client keeps the connection opened by using a small timer that iterates and creates waiting time by calling the `thinktime` action. This makes sure that all the connections stay open during the whole duration of the load test which is important to keep the server loaded with open requests.

First few tests were unsuccessful because the client instance could not generate more than 1000 requests in total. As the load phase from the Tsung configuration file defines, the client instance should generate 10 requests per second for 300 seconds but the tests would stop generation new requests after around 100 seconds. The reason for this was that the system-wide resource limit was being reached on the client instance. Each new open connection is a new open file in the operating system and, by default, the limit for simultaneous open files was 1024 on the client instance. By editing the file `/etc/security/limits.conf` and increasing these limits, this issue was resolved.

8.5 Phoenix prototype tests

The following Tsung XML file is used for load testing the Phoenix prototype application:

```
<!DOCTYPE tsung SYSTEM "/usr/local/Cellar/tsung/1.6.0/share/
   tsung/tsung-1.0.dtd">
<tsung loglevel="debug" version="1.0">
  <clients>
    <client host="localhost" use_controller_vm="true"
       maxusers="30000" />
  </clients>

  <servers>
    <server host="35.162.132.55" port="4000" type="tcp" />
  </servers>

  <load>
    <arrivalphase phase="1" duration="300" unit="second">
```

```xml
      <users maxnumber="30000" arrivalrate="10" unit="second"/>
    </arrivalphase>
  </load>

  <options>
    <option name="ports_range" min="1025" max="65535"/>
  </options>

  <sessions>
    <session name="websocket" probability="100" type="ts_websocket">
      <request>
        <websocket type="connect" path="/socket/websocket"></websocket>
      </request>

      <request>
        <websocket type="message" frame="text">
          {"topic":"chat","event":"phx_join","payload":{},"ref":"1"}
        </websocket>
      </request>

      <for var="i" from="1" to="100" incr="1">
        <thinktime value="25"/>
        <request>
          <websocket ack="no_ack" type="message">{"topic":"phoenix","event":"heartbeat","payload":{},"ref":"2"}</websocket>
        </request>
      </for>
    </session>
  </sessions>
</tsung>
```

The client and server configurations are identical as in the Rails test except that the Phoenix application is running on the port 4000 instead of 3000. The load scenario is also the same, where one load phase is used then generates 10 users per second for 300 seconds. The connection path is modified because the Phoenix server accepts WebSocket connection on the `/socket/websocket` URL instead the `/cable` URL. The format of the message is structured in the way that it requests to join the topic `chat` on the server. The same `thinktime` action timer is used to keep all the connections open during the whole duration of the load test. One additional WebSocket message is sent every 25 seconds to keep the connection opened. It is called a heartbeat and unless it is sent from the client every 30 seconds, the server will declare the client as inactive and will terminate the connection. This last part was not needed in the previous load testing configuration because Rails uses a different strategy to keep the connections alive. Rails is sending these heartbeat messages from the server to determine if the client is still alive. So in both solutions, the purpose of the messages are identical, only the direction of the checks are reversed. For the purpose of this load testing, it does not matter who is responsible for sending these messages just that the behaviors are as similar as possible so that the results are comparable.

8.6 Results

Figure 8.2 shows the first successful test of the Rails prototype application which resulted in server accepting only 1254 connections.

This could indicate that the server instance was overloaded and all the resources exhausted with these 1254 requests and that it could not accept new requests. But the analysis of the server instance logs revealed that the server resources were not depleted and only 30% of the resources were being used but the server did not accept new requests nevertheless.

The error rate for the `error_connect_etimeout` which was quite high indicates that the server timeouts on requests (shown in figure 8.3). Even though the server had resources like CPU and memory to handle these requests, they were still waiting in a queue for the server to accept them. The

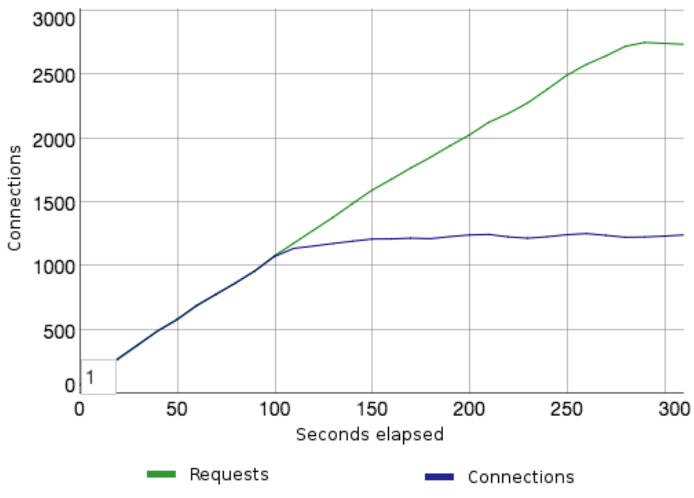


Figure 8.2: Connections rates - Rails prototype (10 req/s)

problem here was that Rails was running in only one instance on the server. The instance can accept only that many requests before moving them to the queue. The solution here was to horizontally scale the solution by running the application in multiple instances. As each Rails instance takes around 200MB of memory, four instances can be run in parallel on the 1GB server instance. The same test was repeated on four running instances and now all generated requests were handled without any issues with queuing or timeout errors.

After the server solution has been horizontally scaled the repeated test provided better results (shown in figure 8.4.) Out of 2951 generated requests, all 2951 were accepted while the server resources were at 70% CPU capacity. The application was fully functional and responsive despite having 2951 open connection. The responsiveness was tested by opening the app in the browser and submitting a chat message which was then broadcasted to all 2952 connected users. Total time for submitting and receiving the WebSocket update with the message was 224 ms which can be considered a good responsive behavior.

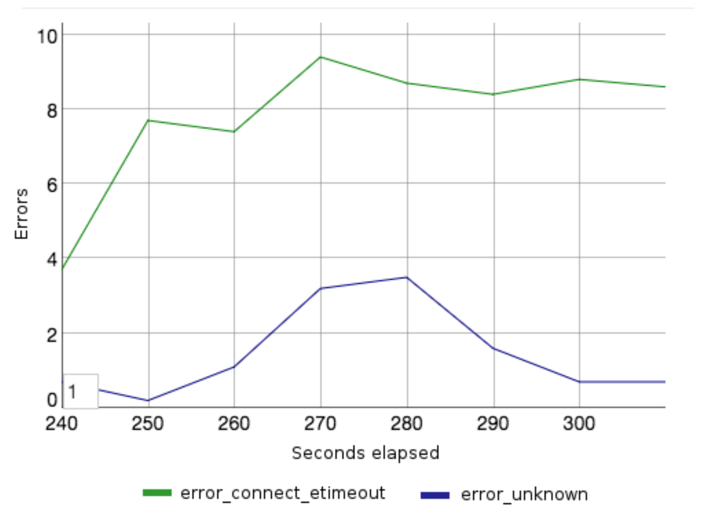


Figure 8.3: Error rates - Rails prototype (10 req/s)

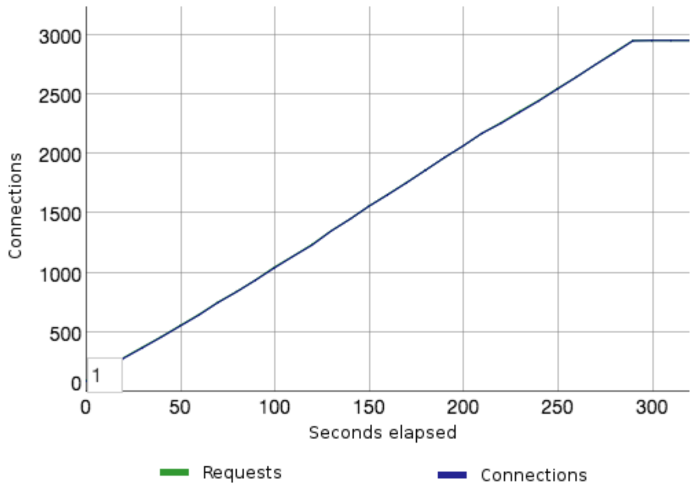


Figure 8.4: Connections rates - Rails prototype (10 req/s)

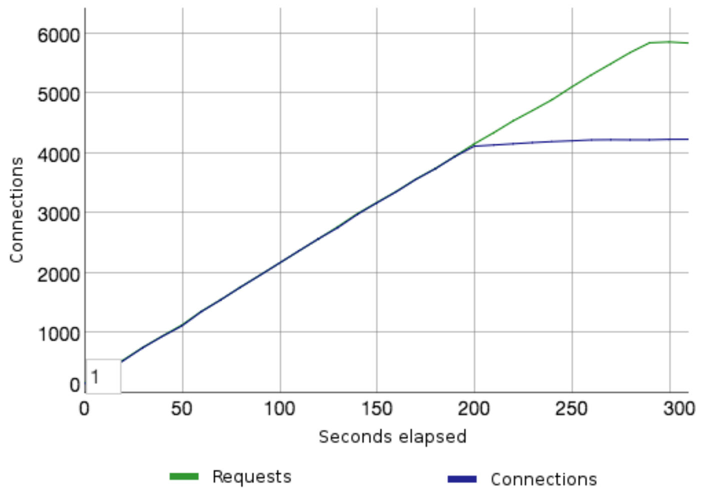


Figure 8.5: Connections rates - Rails prototype (20 req/s)

Next test doubled the requests per second from 10 to 20 which give a total number of 6000 requests in 300 seconds. The results in the figure 8.5 show that out of 5944 generated requests only 4238 were successfully connected.

The server resources were overloaded as the CPU was at 100% usage after around 200 seconds of the test. After that, the server stopped accepting new requests because it needed the resources to keep the existing ones alive. The application was still functional and accessible but not responsive that much. The same test was made, sending the message from the browser. It took 1,533 seconds for the server to accept the request, broadcast to all 4239 connected users and for the message to be received back by the browser which is not considered responsive enough behavior for pleasant usage of the application. Furthermore, the high server load caused an increased error rate in accepting requests. A total of 99 errors happened with the highest rate of 2,2 errors per second (shown in the figure 8.6).

The Phoenix prototype, on the other hand, did not have the issues with accepting the limited amount of connection caused by the shortage in the number of instances like Rails prototype had. Erlang is a fully concurrent

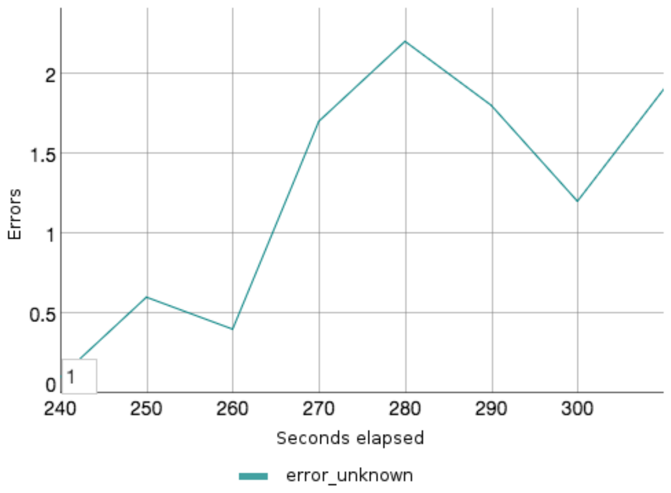


Figure 8.6: Error rates - Rails prototype (20 req/s)

language where only one instance of the application has to be run which is then scaled internally by the distributed multi-process system.

The first load test generated 2963 requests and all 2963 were accepted (shown in figure 8.7). The server resources were only at 20% CPU capacity (compared to 70% for the Rails prototype). The application was also fully functional and responsive and the test chat message was broadcasted to all 2963 connected users in 210 ms which can be considered a good responsive behavior.

The next test doubled the requests per second from 10 to 20 which gives a total number of 6000 requests. For the Rails prototype that was over the capabilities of the solution as only 4238 connections were accepted. The Phoenix had no issues with this test load and out of 5915 generated requests, all 5915 were successfully connected (shown in figure 8.8). Despite having 5915 connected users, the application was fully functional and responsive. The test chat message was broadcasted to all 5915 connected users in 834 ms which can still be considered a good responsive behavior.

Next test doubled the requests per second from 20 to 40 which gives

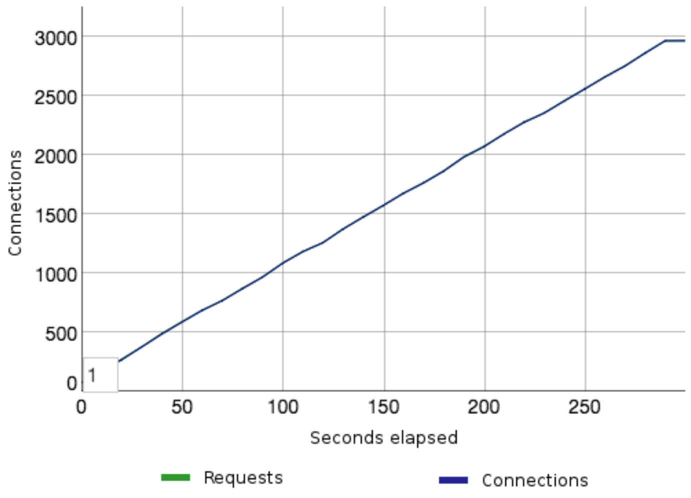


Figure 8.7: Connections rates - Phoenix prototype (10 req/s)

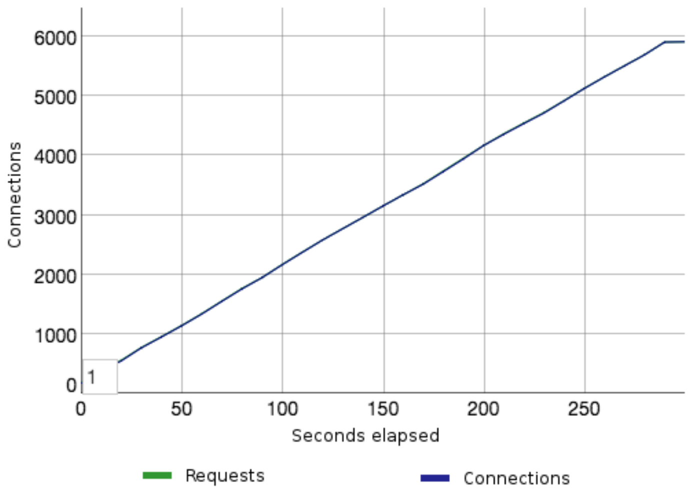


Figure 8.8: Connections rates - Phoenix prototype (20 req/s)

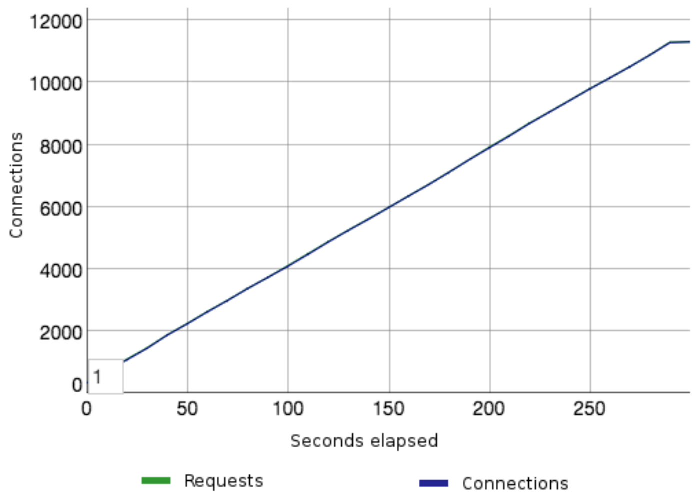


Figure 8.9: Connections rates - Phoenix prototype (40 req/s)

a total number of 12000 requests in 300 seconds. Out of 11314 generated requests, 11313 were successfully connected and only 1 request was unsuccessful (shown in figure 8.9). Despite handling 3 times more requests than the Rails prototype application, the server resources were still not exhausted and CPU load was at 60% which means that the server could accept even more requests.

The last test doubled the number of requests again, from 40 to 80 per second which give a total number of 24000 requests in 300 seconds. Out of 23215 generated requests, 20686 were successfully accepted by the server (shown in figure 8.10).

This load test managed to exhaust all the resources and overload the server CPU after around 220 seconds into the test. The application was still functional and accessible but not responsive that much. It took 5,692 seconds for the server to accept the request, broadcast the message to all 20686 connected users and for the browser to receive and show the message. After the moment that the server got overloaded erroneous responses started happening. A total of 536 errors happened with the highest rate of 12,4

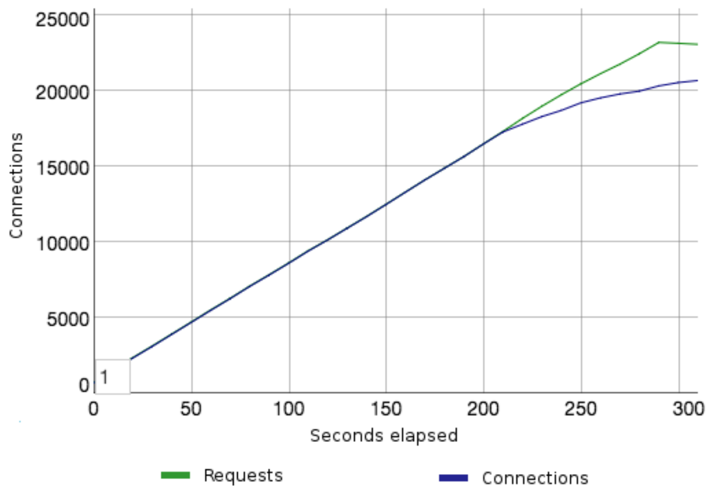


Figure 8.10: Connections rates - Phoenix prototype (80 req/s)

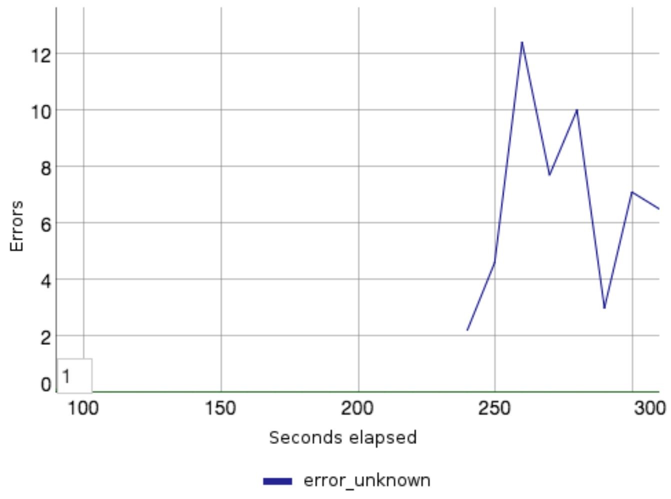


Figure 8.11: Error rates - Phoenix prototype (80 req/s)

errors per second (shown in figure 8.11).

8.7 Conclusions

In the table 8.2 all measured statistical data from the prototype application load tests are shown and compared. Rows with ”R” indicate the Rails prototype application and rows with ”P” indicate the Phoenix prototype application results.

The tests show that the Phoenix prototype managed to accept 20686 user connections and the Rails prototype managed to accept only 4238 user connections. Running on identical infrastructure with the same amount of CPU power and memory, the Phoenix prototype is capable of handling 4,88 times more requests which makes it a more scalable solution. Furthermore, the total transferred data is about 2,5 times smaller for the Phoenix prototype compared to the Rails prototype because of how frameworks structure the formats of the messages. It is important to note here that the Phoenix framework, by default, uses a binary protocol for WebSocket messages which has better performance than a text protocol but the Phoenix prototype was forced to use text protocol instead to make it more similar to the Rails prototype. So the Phoenix prototype could perform even better and reduce the data transfer even more when the binary protocol is used.

Together with the improved scalability, the tests show that the availability of the Phoenix solution is also better. By increasing the Mean Time Before Failure (MTBF) the solution is able to work for a longer period of time before failing, thus improving availability. The MTBF was measured as the time before the solution stopped accepting new connections because at that point its real-time feature becomes unavailable for the users wanting to connect to the service. The Rails prototype failed already on the 20 req/s test and the MTBF was 200 seconds. The Phoenix prototype failed only on the last 80 req/s test with MTBF of 220 seconds. To be able to compare these MTBFs they should be measured with same req/s but as the Rails prototype could

Table 8.2: Collected results from distributed load testing of prototypes

		3k	6k	12k	24k
Requests	R	2951	5874	-	-
	P	2963	5915	11314	23215
Connections	R	2951	4238	-	-
	P	2963	5915	11313	20686
Data sent	R	0,76 MB	1,07 MB	-	-
	P	1,86 MB	3,73 MB	7,12 MB	14,33 MB
Data received	R	10,07 MB	16,84 MB	-	-
	P	2,12 MB	4,24 MB	8,08 MB	16,33 MB
Total data	R	10,83 MB	17,91 MB	-	-
	P	3,98 MB	7,97 MB	15,2 MB	30,66 MB
Error count	R	0	99	-	-
	P	0	0	1	546
Error rate	R	0/s	0,33s	-	-
	P	0/s	0/s	0,003/s	1,78/s
Full load response time	R	224ms	1530ms	-	-
	P	210ms	834ms	1194ms	5692ms
MTBF	R	-	200s	- (100s)	- (50s)
	P	-	-	-	220s

not even be run in the 80 req/s test, the MTBFs for the Rails prototype are inferred from the other tests results. As the Rails prototype is able to accept only 4000 requests, the MTBF for the next two tests can be linearly mapped to 100s and 50s. When compared this way, the Phoenix prototype has 4,4 times better MTBF than the Rails prototype thus making the solution more fault tolerant and available.

All the rules about scaling mentioned before in the chapter 4.1 can be applied for both prototypes. They can be vertically and horizontally scaled and thus increase the total number of simultaneous user connections that the solution can handle. To scale vertically a more powerful EC2 machine could be used for the server instance. For example the AWS EC2 t2.2xlarge (mentioned in table 8.1) instance has 8 CPUs and 32GB of memory. Given the 200MB per instance memory allocation, a t2.2xlarge instance could probably run over 100 instances of the Rails prototype application which would then allow accepting much more requests. Similarly to that, the solution could be scaled horizontally by running several t2.2xlarge instances behind a load balancer and increase the total number of requests even more. With enough server power and parallel server instances, hundreds of thousands user connection can be handled. But the fact that the Phoenix prototype can handle almost 5 times more requests using the same environment, makes it a more scalable solution by default. Furthermore, another big advantage of Phoenix over Rails is that Phoenix is concurrent and distributed by default and no instance orchestration inside the same server is needed. The first Rails load tests failed because not enough instances of the application were created and the programmer has to worry about that. In all the Phoenix prototype tests, no additional work was needed for handling concurrency and everything was scaled internally by the language itself.

Chapter 9

Conclusions

Real-time web applications are an inevitable part of the modern WWW stack that allows users to receive information as soon as it becomes available. From full-fledged real-time applications like real-time collaborative editing tools to web applications with real-time components like built-in chat messengers, real-time is ubiquitous in modern web applications. These real-time components enhance the user experience of modern web applications by speeding up the access to new information making it almost instantly available.

From its beginnings in 1989 up to today, the foundation of the WWW has been in the HTTP protocol. This stateless and unidirectional protocol was designed for exchanging static HTML documents using simple request-response cycles between clients and the server. With the emergence of Web 2.0 and especially real-time web applications, new techniques and protocols have been designed to compensate the shortcomings of the HTTP protocol. Comet-like techniques like polling and long polling, streaming, reverse HTTP, BOSH, Bayeux, SSE, and WebSocket simulate real-time behavior by establishing bidirectional communication over the unidirectional HTTP protocol. After analyzing these client technologies and putting them to the tests, the WebSocket protocol shows the best performance results and is superior in every tested category. Compared to polling, long polling and streaming test results, WebSocket shows a data size reduction up to 55% which directly impacts the speed and throughput of the real-time web application.

By establishing bidirectional full-duplex communication between the client and the server, the real-time behavior is simulated and access to new information is almost instant. But in the same time, it also results in increasing the load on the network and server architecture. Server solutions that serve these real-time applications have to cope with this increased load by being scaled horizontally and vertically. Different methods like adding more hardware power to the server and adding more server nodes to the web cluster are used to increase the throughput of the solution and accommodate the increased number of requests. Another important requirement is the fault tolerance of the solution. Because of the increased load, the server solution is more prone to faulty behavior due to software errors. If the solution is not fault-tolerant enough, it might crash and become totally non-operational and unavailable to the users. A high available system is highly fault-tolerant and has an uptime measured in several nine of availability per year.

Both scalability and high availability are very important factors for real-time applications. Real-time server solutions that are handling hundreds of thousands of user requests have to be scalable to cope with high traffic and also fault tolerant to gracefully handle exceptional conditions and keep the service always available and operational. One factor that greatly affects the scalability and fault tolerance is its underlying technology stack and the programming language as its core. Not all programming languages for server solutions are suited for real-time applications and although most provide some basic scalability and high availability features, some are a better fit than others.

The Erlang programming language was designed in 1998 to support massively scalable real-time systems with high requirements on fault tolerance like telecom and banking applications. These high concurrency requirements are achieved by using a powerful set of primitives for creating processes and managing communication among them. However, its application for server solutions has only started lately, after some modern functional based programming languages have been designed that run on the Erlang virtual machine called BEAM like Elixir. Real-time web applications built on top of the Erlang stack should exploit these scalability advantages and perform better in terms of speed, throughput and error rate.

Distributed load testing can be used to measure the performance of real-time web applications. To simulate high loads on the server, a cloud-based testing environment is used where one machine is used to generate requests and the other one serves the server solution. A great distributed load testing automation tool called Tsung is used to orchestrate these tests. After running several tests and comparing results with a similar server solution, the Erlang-based solution could handle almost 5 times more requests with better MTBF while running in the same environment. Also, the error rate was smaller and the response times were better compared to the other solution. All the measurements taken into account, the Erlang-based solution is proven to be more scalable because it can handle more user requests using the same hardware and more fault-tolerant because it reported better MTBF compared to the similar server solution.

The presented solution can be optimized further by using binary protocols, caching mechanisms, and other software improvements. By using the standard scaling techniques like adding more hardware and distributing the load on multiple parallel nodes, a real-time application built on Erlang can be scaled to handle millions of user requests while remaining highly available and operational. This makes Erlang a viable solution for developing server solution for real-time web applications. The future of the real-time web is greatly interlaced with the future of Erlang-based web solution stack.

Bibliography

[1] T. J. Berners-Lee, R. Cailliau, J. F. Groff, The world-wide web, Computer Networks and ISDN Systems 25 (1992) 454–459.

[2] T. J. Berners-Lee, R. Cailliau, J. F. Groff, B. Pollermann, World-wide web: The information universe, Internet Research 2 (1) (1992) 52–58.

[3] K. Jeffay, F. D. Smith, F. Hernandez-Campos, Tracking the evolution of web traffic: 1995-2003, Proceedings of the 11th IEEE/ACM International Symposium on MASCOTS (2003) 16–25.

[4] T. J. Berners-Lee, M. Fischetti, Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor, Barnes and Noble, 1999.

[5] B. Sawyer, D. Greely, Creating GeoCities Websites, Muska and Lipman Publishing, 1999.

[6] D. DiNucci, Fragmented future, Print 32 (1999) 221.

[7] T. O'Reilly, What is web 2.0. (2005).
URL `http://www.oreilly.com/pub/a//web2/archive/what-is-web-20.html`

[8] G. Cormode, B. Krishnamurthy, Key differences between web 1.0 and web 2.0 (2008), First Monday 13 (6).

[9] E. K. Gill, How can we measure the influence of the blogosphere (2004), Proceedings of the WWW 2004 Conference.

[10] J. Nielsen, Usability Engineering, Morgan Kaufmann Publishers, 1993.

[11] J. Stankovic, Real-time computing (1992), BYTE 17 (8).

[12] D. R. Herrick, Google this!: using google apps for collaboration and productivity, Proceedings of the 37th annual ACM SIGUCCS (2009) 55–64.

[13] S. Dekeyser, R. Watson, Extending google docs to collaborate on research papers (2006), Technical report, University of Southern Queensland, Australia.

[14] S. Greenberg, D. Marwood, Real time groupware as a distributed system: concurrency control and its effect on the interface, In Proceedings of the ACM conference on CSCW (1994) 207–217.

[15] C. Gutwin, M. Lippold, N. Graham, Real-time groupware in the browser: Testing the performance of web-based networking, Computer Supported Cooperative Work (2011) 167–176.

[16] B. Zimmer, A. Kerren, Harnessing webgl and websockets for a web-based collaborative graph exploration tool, Integrated Communications, Worldwide events (2015) 23–26.

[17] Y. Zhangling, D. Mao, A real-time group communication architecture based on websocket, International Journal of Computer and Communication Engineering 1 (4) (2012) 409–411.

[18] S. Rakhunde, Real time data communication over full duplex network using websocket, IOSR Journal of Computer Science (2014) 15–19.

[19] V. Pimentel, B. Nickerson, Communicating and displaying real-time data with websocket, IEEE Internet Computing 16 (4) (2012) 45–53.

[20] W. Zhang, R. Stoll, N. Stoll, K. Thurow, An mhealth monitoring system for telemedicine based on websocket wireless communication, Journal of networks 8 (4) (2013) 955 – 962.

[21] N. Kulkarni, T. Eltaieb, Video streaming over full duplex network using websocket and its performance evaluation, Journal of Multidisciplinary Engineering Science and Technology 2 (4) (2015) 589 – 592.

[22] L. Srinivasan, J. Scharnagl, K. Schilling, Analysis of websockets as the new age protocol for remote robot tele-operation, 3rd IFAC Symposium on Telematics Applications (2013) 83 – 88.

[23] B. Chen, Z. Xu, A framework for browser-based multiplayer online games using webgl and websocket, International Conference on Multimedia Technology (2011) 471 – 474.

[24] S. Arora, J. Maini, P. Mallick, Efficient e-learning management system through web socket, International Conference on Computing for Sustainable Global Development (2016) 509 – 512.

[25] T. J. Berners-Lee, The original http as defined in 1991 (1991).
URL `https://www.w3.org/Protocols/HTTP/AsImplemented.html`

[26] T. J. Berners-Lee, R. Fielding, H. Frystyk, Hypertext transfer protocol – http/1.0 (rfc1945) (1996).
URL `https://www.rfc-editor.org/rfc/rfc1945.txt`

[27] T. J. Berners-Lee, R. Fielding, H. Frystyk, J. Gettys, J. Mogul, Hypertext transfer protocol – http/1.1 (rfc2068) (1997).
URL `https://www.rfc-editor.org/rfc/rfc2068.txt`

[28] T. J. Berners-Lee, R. Fielding, H. Frystyk, J. Gettys, J. Mogul, L. Masinter, P. Leach, Hypertext transfer protocol – http/1.1 (rfc2616) (1999).
URL `https://www.rfc-editor.org/rfc/rfc2616.txt`

[29] R. Fielding, J. Reschke, Hypertext transfer protocol (http/1.1): Message syntax and routing (rfc7230) (2014).
URL `https://www.rfc-editor.org/rfc/rfc7230.txt`

[30] J. Postel, Transmission control protocol (1981).
URL `https://www.rfc-editor.org/rfc/rfc793.txt`

[31] J. Postel, User datagram protocol (1980).
URL `https://www.rfc-editor.org/rfc/rfc768.txt`

[32] B. A. Forouzan, TCP/IP Protocol Suite, McGraw-Hill, 2002.

[33] T. J. Berners-Lee, R. Fielding, L. Masinter, Uniform resource identifier (uri): Generic syntax (2005).
URL `https://www.rfc-editor.org/rfc/rfc3986.txt`

[34] curl.1 the man page (accessed: September 2017).
URL `https://curl.haxx.se/docs/manpage.html`

[35] H. Parmar, M. Thornburgh, Adobe's real time messaging protocol (2012).
URL `http://wwwimages.adobe.com/content/dam/Adobe/en/devnet/rtmp/pdf/rtmp_specification_1.0.pdf`

[36] W. Sanders, Learning Flash Media Server 3, O'Reilly, 2008.

[37] Historical yearly trends in the usage of client-side programming languages for websites (accessed: September 2017).
URL `https://w3techs.com/technologies/history_overview/client_side_language/all/y`

[38] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, J. Cowan, Extensible markup language (xml) 1.1 (2006).
URL `https://www.w3.org/TR/2006/REC-xml11-20060816/`

[39] T. Bray, The javascript object notation (json) data interchange format (2014).
URL `https://www.rfc-editor.org/rfc/rfc7159.txt`

[40] Xmlhttprequest (accessed: September 2017).
URL `https://developer.mozilla.org/en/docs/Web/API/XMLHttpRequest`

[41] Xmlhttprequest browser support (accessed: September 2017).
URL `https://caniuse.com/#feat=xhr2`

[42] A. Barth, The web origin concept (2011).
URL `https://www.rfc-editor.org/rfc/rfc6454.txt`

[43] Apache server keepalivetimeout config (accessed: September 2017).
URL `https://httpd.apache.org/docs/2.4/mod/core.html#keepalivetimeout`

[44] N. E. Sit, Reverse http tunneling for firewall traversal (2000), Master thesis at Massachusetts Institute of Technology.

[45] Reverse http draft (2009).
URL `https://tools.ietf.org/html/draft-lentczner-rhttp-00`

[46] Reverse http specifications (2009).
URL `http://reversehttp.net/reverse-http-spec.html`

[47] I. Paterson, D. Smith, P. Saint-Andre, J. Moffitt, L. Stout, W. Tilanus, Xep-0124: Bidirectional-streams over synchronous http (bosh) (2009).
URL `https://xmpp.org/extensions/xep-0124.html`

[48] M. Laine, K. Saila, Performance evaluation of xmpp on the web (2012), Master thesis at Massachusetts Institute of Technology.
URL `https://pdfs.semanticscholar.org/23f8/5450ab0cec26bd2e72ccaa09704682d79dcd.pdf`

[49] A. Russell, G. Wilkins, D. Davis, M. Nesbitt, The bayeux protocol specification 1.0 (2007).
URL `https://docs.cometd.org/current/reference/#_bayeux`

[50] T. J. Berners-Lee, R. Fielding, H. Frystyk, J. Gettys, J. Mogul, L. Masinter, P. Leach, Hypertext transfer protocol – http/1.1 (rfc2616) - connections - practical considerations (1999).
URL `https://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html#sec8.1.4`

[51] E. Kahn, J. Riemer, L. Lechner, Xvsmp/bayeux: A protocol for scalable space based computing in the web, 2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises 0 (2007) 68–73.

[52] I. Hickson, Server-sent events (2015).
URL `https://www.w3.org/TR/eventsource/`

[53] Eventsource (accessed: September 2017).
URL `https://developer.mozilla.org/en-US/docs/Web/API/EventSource`

[54] Server-sent events browser support (accessed: September 2017).
URL `http://caniuse.com/#feat=eventsource`

[55] R. N. Darwish, M. I. Abdelwahab, Impact of implementing http/2 in web services, International Journal of Computer Applications 147 (9) (2016) 27–32.

[56] M. Belshe, R. Peon, M. Thomson, Hypertext transfer protocol version 2 (http/2) (2015).
URL `https://www.rfc-editor.org/rfc/rfc7540.txt`

[57] R. Peon, H. Ruellan, Hpack: Header compression for http/2 (2015).
URL `https://www.rfc-editor.org/rfc/rfc7541.txt`

[58] M. A. Abdillahi, U. Dossetov, A. Saqib, Performance evaluation of http/2 in modern web and mobile devices, American Journal of Engineering Research 6 (4) (2017) 40–45.

[59] I. Hickson, D. Hyatt, Html5 - tcpconnection (2008).
URL https://www.w3.org/TR/2008/WD-html5-20080610/comms.html#tcpconnection

[60] I. Fette, Google chrome 4 - websocket support (2010).
URL https://blog.chromium.org/2010/01/more-resources-for-developers.html

[61] I. Fette, A. Melnikov, The websocket protocol (rfc6455) (2011).
URL https://www.rfc-editor.org/rfc/rfc6455.txt

[62] I. Hickson, The websocket api (2012).
URL https://www.w3.org/TR/websockets/

[63] Websocket browser support (accessed: September 2017).
URL https://caniuse.com/#feat=websockets

[64] P. Lubbers, F. Greco, Html5 websocket: A quantum leap in scalability for the web (2010).
URL http://www.websocket.org/quantum.html

[65] M. Michael, J. E. Moreira, D. Shiloach, R. W. Wisniewski, Scale-up x scale-out: A case study using nutch/lucene, Parallel and Distributed Processing Symposium (2007) 1–8.

[66] H. Ludwig, A. Keller, A. Dan, R. P. King, R. Franck, Web service level agreement (wsla) language specification (2003).
URL https://www.researchgate.net/profile/Heiko_Ludwig/publication/200827750_Web_Service_Level_Agreement_WSLA_Language_Specification/links/0912f50bcf2dfe836b000000.pdf

[67] Z. Chaczko, V. Mahadevan, S. Aslanzadeh, C. Mcdermid, Availability and load balancing in cloud computing, 2011 International Conference on Computer and Software Modeling 14 (2011) 134–140.

[68] M. Schneider, Self-stabilization, ACM Computing Surveys 25 (1) (1993) 45–67.

[69] E. Marcus, H. Stern, Blueprints for High Availability, Wiley Publishing, 2003.

[70] J. Armstrong, Concurrency oriented programming in erlang (2002).
URL `http://www.rabbitmq.com/resources/armstrong.pdf`

[71] J. Armstrong, Making reliable distributed systems in the presence of software errors, Swedish institute of computer science, 2003.

[72] A. O'Connell, Inside erlang, the rare programming language behind whatsapp's success (2014).
URL `https://www.fastcompany.com/3026758/inside-erlang-the-rare-programming-language-behind-whatsapps-success`

[73] Ruby on rails framework (accessed: September 2017).
URL `http://rubyonrails.org/`

[74] Django framework (accessed: September 2017).
URL `https://www.djangoproject.com/`

[75] Spring framework (accessed: September 2017).
URL `https://spring.io/`

[76] Elixir language (accessed: September 2017).
URL `https://elixir-lang.org/`

[77] Phoenix framework (accessed: September 2017).
URL `http://www.phoenixframework.org/`

[78] A. Orebaugh, G. Ramirez, J. Burke, L. Pesce, J. Wright, G. Morris, Wireshark and Ethereal Network Protocol Analyzer Toolkit, Syngress Publishing, 2007.

[79] Wireshark (accessed: September 2017).
URL https://www.wireshark.org/

[80] R. Wach, Method of performing distributed load testing (2003).
URL https://www.google.com/patents/US20030009544

[81] Goad - load testing tool (accessed: September 2017).
URL https://goad.io/

[82] Artillery - a modern load testing toolkit (accessed: September 2017).
URL https://artillery.io/

[83] Locust - a modern load testing framework (accessed: September 2017).
URL http://locust.io/

[84] Apache jmeter (accessed: September 2017).
URL http://jmeter.apache.org/

[85] Tsung - open-source multi-protocol distributed load testing tool (accessed: September 2017).
URL http://tsung.erlang-projects.org/

[86] Tsung - background (accessed: September 2017).
URL http://tsung.erlang-projects.org/user_manual/introduction.html#tsung-background

[87] Capistrano - remote multi-server automation tool (accessed: September 2017).
URL https://github.com/capistrano/capistrano

[88] Capistrano - remote multi-server automation tool for phoenix (accessed: September 2017).
URL https://github.com/dabit/capistrano-phoenix

[89] Semaphore - hosted continuous integration and deployment service for private and open source projects (accessed: September 2017).
URL https://semaphoreci.com

[90] Codeship - a continuous integration platform in the cloud (accessed: September 2017).
URL https://codeship.com

[91] Heroku - cloud application platform (accessed: September 2017).
URL https://www.heroku.com