CRANFIELD UNIVERSITY

Gabriel Muller

HTML5 WebSocket protocol and its application to distributed computing

SCHOOL OF ENGINEERING

Computational Software Techniques in Engineering

MSc

Academic Year: 2013 - 2014

Supervisor: Mark Stillwell

July 2014

CRANFIELD UNIVERSITY

SCHOOL OF ENGINEERING

Computational Software Techniques in Engineering

MSc

Academic Year: 2013 - 2014

Gabriel Muller

HTML5 WebSocket protocol and its application to distributed computing

Supervisor: Mark Stillwell

July 2014

This thesis is submitted in partial fulfilment of the requirements for the degree of Master of Science

Declaration of Authorship

I, Gabriel L. Muller, declare that this thesis titled, HTML5 WebSocket protocol and its application to distributed computing and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Abstract

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too…

Acknowledgements

I wish to thank my supervisor Mark Stillwell for accepting to work with me on this project. His positive suggestions and willingness to repeadtedly meet and discuss the progress of this thesis have been invaluable.

Naturally, I am also grateful for the continuous support of my family and of my housemates. Always available and willing to provide either advises or distractions.

Lastly I would like to thank my college Léo Unbekannt. Not only did he help me in the very begenning when I was looking for a subject, but also later on through the whole thesis. More importantly our usual lunch conversation rose and confirmed my interest in computer science. Thanks Léo.

Contents

List of Tables

Abbreviations

LAH	**L**ist **A**bbreviations **H**ere

Chapter 1

Introduction

This first chapter ..

1.1 Javascript

In order to keep up with the evolution and continuous growth of the Internet, web technologies have been undergoing significant upgrades. Since 2007, the World Wide Web Consortium (W3C) has been working on a major update of the core language of the web that renders and displays all web contents. This is known as the 5th revision of Hyper Text Markup Language (HTML5). However the slow performance of JavaScript in performing dynamic operations is a serious limiting factor to wider use. Improving the efficiency of JavaScript is an active field of research.

1.2 Client server communications

Client server communication have been built around the HTTP protocol. CHANGE INTRO ..

1.2.1 HTTP protocol

The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity meta information, and possible entity-body content. [1]

Because HTTP was not designed for real time communication several workarounds have been developped over the years to over come the so called page by page model. These techniques are nicely resumed in Eliot Step master thesis [2].

1.2.2 Page by page model

Since HTTP's release in 1991, client server communication have undergone continuous upgrades. In the early twenties, most web pages were static. As a consequence, the client server communication were rather limited. Typically, the client would send from time to time a request to the server, the server would then answer back. All communication would then stop until the client triggered a new event.

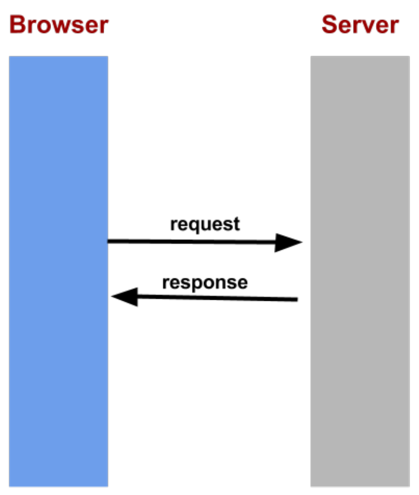


Figure 1.1: Client server communication

The notion of dynamic web appeared in 2005 with the apparition of technologies like Comet. Peter Lubbers describes it as the Headache 2.0 in his article *A quantum leap in scalability for the web* [3].

1.2.3 1.2.3 Polling

Polling was the first attempt for real-time communication. Instead of waiting for the client to manually ask for a page update, the browser was sends regular HTTP GET requests to the server. This technique could be efficient if the exact interval of update on the server side was known.

However real time information are unpredictable and in high updates rate situation like for example stock prices, news reports or tickets sales the response could be stale by the time the browser renders the page [3].

Also in low updates rate situation even if no data is available, the server will send an empty response. Resulting in a large amount of unnecessary connections beeing established which over time and with the clients increase leads to decreased overall network throughput [2].

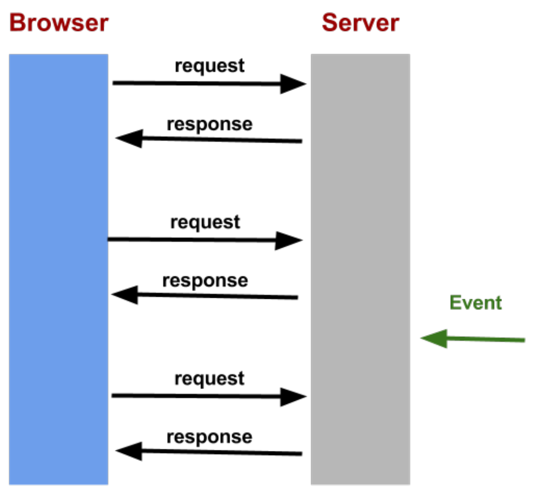


FIGURE 1.2: polling

1.2.4 Long polling

Long polling is based on Comet technologies and is slight step further toward server events and real time communication. Comet began to be popular in web browser around 2007, it is a family of web techniques that allows the server to hold an HTTP request open for prolonged periods of time.

Long-polling is similar to polling, except that the server keeps the HTTP request open if data is not immediately available. The server determines how long to keep the request open, also known as a hanging GET". If new data is received within the time interval, a response containing the data is sent to the client and the connection is closed. If new data is not received within the time period, the server will respond with a notification to terminate the open request and close the connection. After the client browser receives the response, it will create another XHR object request to handle the next event, therefore always keeping a new long-polling request open for new events. This results in the server constantly responding with new data as soon as it is made available [2].

However, in situations with high-message volume, long- polling does not provide increased performance benefits over regular polling. Performance could actually be decreased if long-polling requests turn into continuous, unthrottled loops of regular polling requests.

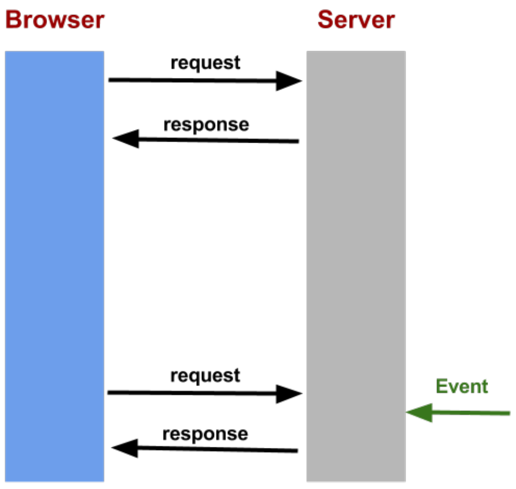


FIGURE 1.3: Long polling

1.2.5 Streaming

Streaming is based on a persistent HTTP connection. The communication still begins with a request from the browser, the difference is in the response. The server never signals the browser its message is finished. This way the connection is kept open and ready to deliver futher data. [2].

Wouldn't it be because of proxies, streaming would be a perfectly adapted for real time communication. Because streaming is done over HTTP, proxy server may choose to buffer server responses and thus increasing greatly the latency of the message delivery. Therefore in case a proxy is detected most Comet-like solution fall back to long polling [2].

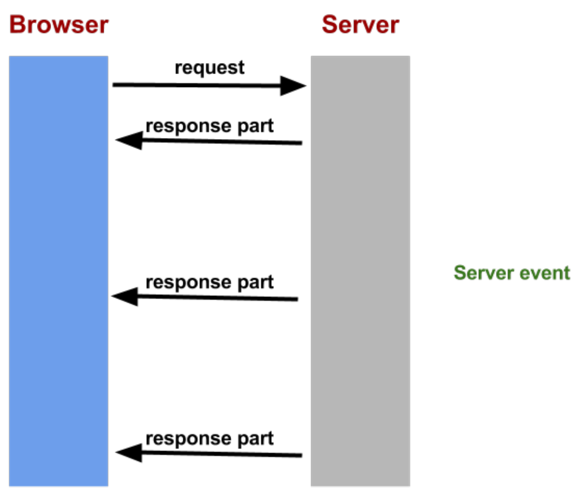


Figure 1.4: Streaming

1.2.6 Current technologies in browser

At the moment, comet technologies are still the most popular way of communication between browsers and servers. The technique has been improved to the point where it perfectly fakes server sent event. Comet technologies can be seen as a wonderful hack to reach real time communication. However little can be done to improve the latency. Comet technologies resolve around HTTP and carry its overhead

The total overhead from the HTTP request and response header is at least 871 bytes without containing any data. In comparaison, a small payload is 20 bytes. Contempory application like on-line games can not be built on a technology waisting ressources equivalent to 40 messages every time informations are exchanged [2]. Therefore a brand new protocol has been developed: WebSocket.

1.3 WebSocket protocol

The creation of the WebSocket protocol marks the begenning of the Living web. It is often referred to as the first major upgrade in the history of web communications. As the Web itself originally did, WebSocket enables entirely new kinds of applications. Daily, new products are designed to stay permanently connected to the web. Websocket is the language enabling this revolution.

The official Request For Comments [4] (RFC) describes the WebSocket protocol as follows.

The WebSocket Protocol enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code. The security model used for this is the origin-based security model commonly used by web browsers. The protocol consists of an opening handshake followed by basic message framing, layered over TCP. The goal of this technology is to provide a mechanism for browser-based applications that need two-way communication with servers that does not rely on opening multiple HTTP connections.

This section is structured to follow step by step the creation of a WebSocket communication channel. The first subsection is about the initialization of this channel also called handshake. The second about the transport layer. After what the messages frame anatomy will be discussed and to finish WebSocket's behavior toward firewalls will be briefly described.

1.3.1 The WebSocket handshake

The Websocket protocol was to be released in an already existing web infrastrucure. Therefore it has been designed to be backward-compatible. Before a Websocket communication can start, a HTTP connection must be initiated. The browser sends an Upgrade header to the server to inform him he wants to start a WebSocket connection. Switching from the HTTP protocol to the WebSocket protocol is referred to as a handshake [4].

```
GET ws://websocket.example.com/ HTTP/1.1
Origin: http://example.com
Connection: Upgrade
Host: websocket.example.com
Upgrade: websocket
```

If the server supports the WebSocket protocol, it sends the following header in response.

```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Wed, 5 May 2014 04:04:22 GMT
Connection: Upgrade
Upgrade: WebSocket
```

After the completion of the handshake the WebSocket connection is active and either the client or the server can send data. The data is contained in frames, each frame is prefixed with a 4-12 bytes to ensure the message can be reconstructed. Once the server and the browser have agreed on begenning a WebSocket communication. A first request is made to begin an ethernet communication followed by a request to make an TCP / IP communication.

1.3.2 Transport layer protocol

The internet is based on two transport layer protocol, the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). Both use the network layer service provided by the internet protocol (IP).

TCP

TCP is a reliable transmission protocol. The data is buffered bytes by bytes in segments and transmitted according to specific timers. This flow control ensures the consistancy of the data. TCP is said to be a stream oriented because the data is sent in independent segments.

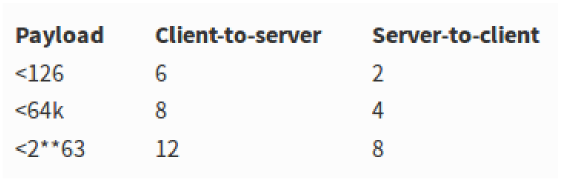
UDP

UDP is an unreliable but fast transmission protocol. The protocol offers no guarentees the data will be delivered in its integrality nor duplicated. It works on a best effort strategy with no flow control. Each segments are received independently, it is a message oriented protocol.

Websocket is build over TCP because of its realiability. Browser enabled games are the perfect use case example of WebSockets. They require low latency even though the rate of update is rather high. To achieve a low latency, the communication protocol must make sure not to drop any packets. Otherwise, the exhange takes two times longer.
As can be inferred from the 2 previous subsections, the websockets protocol is depedant of quite a few other protocols. Namely HTTP to initialize the communication then ethernet, TCP/IP and finaly TLS in case a secure connections is required. The next subsections looks more in details the anatomy of the frames sends over the web when using the websocket protocol.

1.3.3 The WebSocket frame anatomy

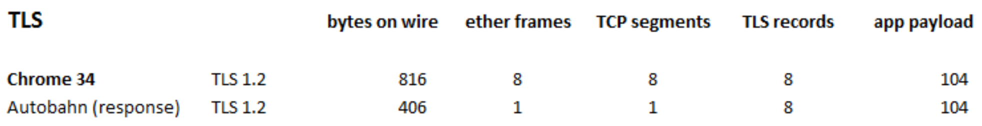
The study conducted by Tobias Oberstein [5] looks into the overheads of websockets. As a matter of fact the overhead induced purely by WebSockets is extremely low. As can be seen in the figure 1.5, depending on the size of the payload the overhead varies between 8 and 20 bytes.



Payload	Client-to-server	Server-to-client
<126	6	2
<64k	8	4
<2**63	12	8

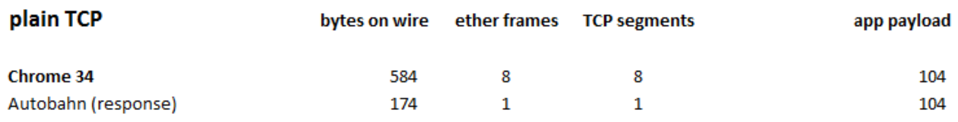
FIGURE 1.5: Frame overhead [5]

However, as pointed out in the article efficiency is lost on all other layers the websocket protocol relies on. Figure 1.6 and 1.7 show respectively the overhead induced by pure tcp/ip and tls protocols.



TLS		bytes on wire	ether frames	TCP segments	TLS records	app payload
Chrome 34	TLS 1.2	816	8	8	8	104
Autobahn (response)	TLS 1.2	406	1	1	8	104

FIGURE 1.6: TLS overhead [5]

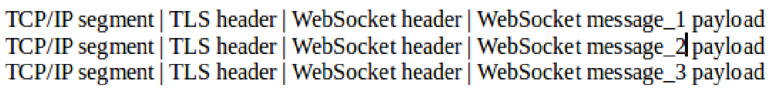


plain TCP	bytes on wire	ether frames	TCP segments	app payload
Chrome 34	584	8	8	104
Autobahn (response)	174	1	1	104

FIGURE 1.7: TCP overhead [5]

In this example, the payloads *Hello world* is only therteen bytes. In comparaison, the ethernet, tcp/ip and tls protocols each use height bytes. The conclusion of this article is to warn programmers about the size of the payloads to make sure ethernet, tcp/ip and tls protocols don't dwarf the overhead of WebSocket itself. In case small payloads can not be avoided a possible solution is to serialize the messages in order to batch them in one single WebSocket message.

So instead of sending the each messages using the WebSocket protocol like it is done in figure x. The individual messages are put in a queue and batched in a single Websocket message like in figure y.



TCP/IP segment | TLS header | WebSocket header | WebSocket message_1 payload
TCP/IP segment | TLS header | WebSocket header | WebSocket message_2 payload
TCP/IP segment | TLS header | WebSocket header | WebSocket message_3 payload

FIGURE 1.8: WebSocket messages sent individualy [5]

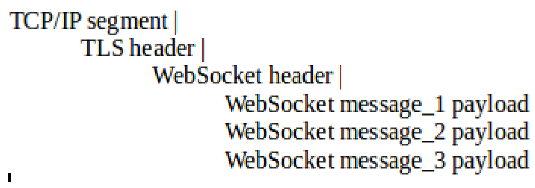

TCP/IP segment |
TLS header |
WebSocket header |
WebSocket message_1 payload
WebSocket message_2 payload
WebSocket message_3 payload

FIGURE 1.9: Batched WebSocket messages [5]

Never the less, WebSockets carry way less overheads then comet technologies does. Another advantage of WebSocket its interaction with proxies.

1.3.4 Proxies

Proxy servers are set up between a private network and the Internet. They act like an intermediary providing content caching, security and content filtering.

When a Websocket server detects the presence of a proxy server, it automatically sets up a tunnel to pass through the proxy. The tunnel is established by issuing an HTTP CONNECT statement to the proxy server, which requests for the proxy server to open a TCP/IP connection to a specific host and port. Once the tunnel is set up, communication can flow unimpeded through the proxy.

LINK

1.4 Scope of the thesis

1.5 Thesis structure

Chapter 2

Literrature review

This second chapter is composed of two parts. The first one is ..

2.1 Implementation

This first chapter looks into studies done around WebSockets. The first section will study WebSocket server implementation and the second one heterogeous implementation.

2.1.1 WebSocket server implementation

Language Selection

Choosing a language for a project is often a compromise between the programmer development background and the necessity of the application. Furthermore, WebSocket servers can be developped in almost any languages.

This subsection does not aim at giving a comprehensive comparaison of all existing WebSocket friendly languages. Also node.js seems to be the perfect environment for this study, therefore other languages will deliberatly be left apart.

Node.js was specially invented to create real-time websites with push capabilities [6]. Most languages run parallel tasks by using threads but threads are memory expensive. Node.js is fundamentally different, it runs as a single non-blocking and event-driven loop by using asynchronous call back loops [7]. For this reasons, compared to other languages, Node.js performs significantly better in highly concurrent environment.

Node.js has many real-time engines. The next step is to carefully make a choice between ws, Socket.io and Engine.io. WebSocket implementation selection

Deniz Ozger article for medium.com [8]is a comprehensive study of node.js real-time engines. Ws is a pure WebSocket implementation, therefore it is interesting for testing purpose but seldom used in real life projects. The main drawback is the communication may not work in case the browser does not support WebSockets.

Socket.io has some appreciable features namely it's connection procedure. First tries to connect to a server via WebSocket after what it downgrades until it founds a suitable protocol. Moreover it tries to reconnect sockets when connections fail. Engine.io is a lower library of Socket.io. The connection procedure is the opposite to Socket.io though. It first establishes a long polling connection and only later tries to upgrade it to a better transport protocol. Therefore it is more reliable because it establishes less connection.

LINK

2.1.2 Heterogeneous implementation with OpenCL

As suggest John Stone paper's title OpenCL: A parallel programming standard for heterogeneous computing systems [9] OpenCL is unanimously considered as the refetence for heterogenous computing.

Historically, the first technology to take advantage of the massive parallel nature of GPUs was Open Graphic Library (OpenGL). OpenGL is an application programming interface (API) for rendering 2D and 3D vector graphics. Through the insertion of little pieces of C-like codes in shader, developpers soon realized graphic processing units (GPUs) could also be used for general programming. This became known as General Purpose computation on GPUs (GPGPU) [9].

However, shadders can only be modified so much. As the need for more complex applications arose Apple proposed the Khronos Group to develop a more general framework: OpenCL. OpenCL is a low-level API accelerating applications with task-parallel or data-parallel computations in a heterogeneous computing environment. Indeed OpenCL allows not only the usage of CPUs but also any processing devices like GPUs, DSPs, accelerators and so on [9]. If generally on desktop the diversity of processing devices is quite low, it is the opposite for mobile. Embedded systems for real-time multimedia journal published a paper [10] highlining the advantages of using OpenCl in mobile browser.

OpenCL doesn't guarantee a particular kernel will achieve peak performance on different architectures. The nature of the underlying hardware may induce different programming strategies. Multi-core CPU architecture is definitely the more popular. But the recent specification published by Khronos to take GPU computing to the web is bound to raise programmers interest toward GPUs architecture [**?**].

CPUs architecture

Modern CPUs are typically composed of a few high-frequency processor cores. CPUs perform well for a wide variety of applications, but they are optimal for latency sensitive workloads with minimal parallelism. However, to increase performance during arithmetic and multimedia workloads, many CPUs also incorporate small scale use of single-instruction multiple-data (SIMD).

GPUs architecture

Contemporary GPUs are composed of hundreds of processing units running at low frequency.

As a result GPUs are able to execute tens of thousands of threads. It is this ability which makes them so much more effective then CPUs in a highly parallel environment. Some research even claims a speedup in the order of 200x over JavaScript. [10]

The GPU processing units are typically organized in SIMD clusters controlled by single instruction decoders, with shared access to fast on-chip caches and shared memories. Massively parallel arithmetic-heavy hardware design enables GPUs to achieve single-precision floating point arithmetic rates approaching 2 trillions of instructions per second (TFLOPS). [9]

Although GPUs are powerful computing devices, currently they still often require management by a host CPU. Fortunately OpenCL is designed to be used in heterogeneous environment. It abstracts CPUs and GPUs as compute devices. After which, applications can query device attributes to determine the properties of the available compute units and memory systems. [9]

All the same, even if OpenCL's API hides the trickiest part of parallel programming a good understanding of the underlying memory model leads to more efficient codding. Along with general advises on how to build an OpenCL cluster, details about the memory model are given here [11].

Platform model

CPU and GPU are called compute devices. A single host regroups one or more compute devices and has its own memory. Each compute device is composed of one or more cores also called compute units. Each compute unit has its own memory and is divided into one or more single-instruction multiple-data (SIMD) threads or processing elements with its own memory. [11]

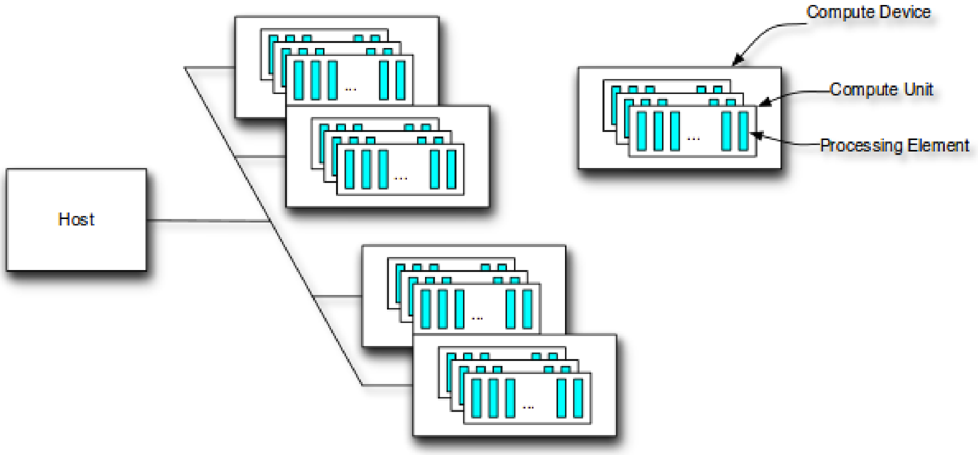


FIGURE 2.1: Plateform model [11]

Memory model

OpenCL defines 4 types of memory spaces within a compute device. A large high-latency global memory corresponding to the device RAM. This is a none cached memory where the data is stored and is available to all items. A small low-latency read-only constant memory which is cached. A shared local memory accessible from multiple processing elements within the same compute unit and a private memory accessible within each processing element. This last type of memory is very fast and is the register of the items. The last and least used kind of memory is texture memory, which is similar to global memory but is cached. [11]

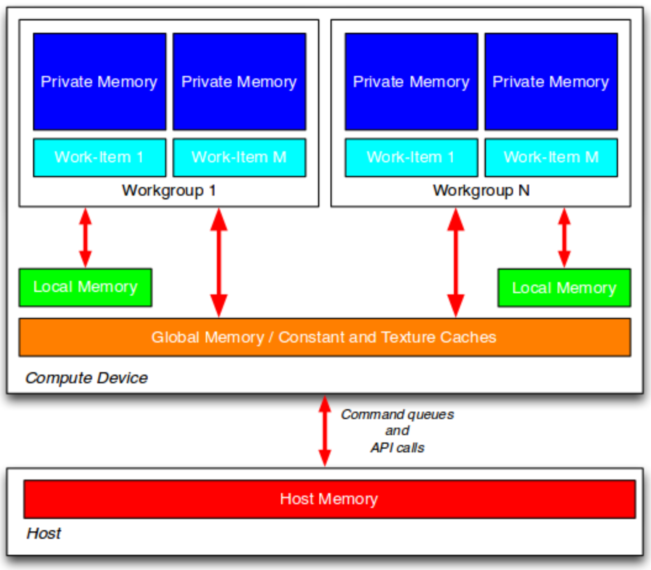


FIGURE 2.2: Memory model [11]

In conclusion, OpenCL provides a fairly easy way to write parallel code but to reach an optimal performance / memory access trade off programmers must choose carefully in where to save their variables in memory space.

Global and local IDs

Finally, at an even lower level, work-items are scheduled in work groups. This is the smallest unit of parallelism on a device. Individual work-items in a work group start together at the same program address, but they have their own address counter and register state and are therefore free to branch and execute independently. [11]

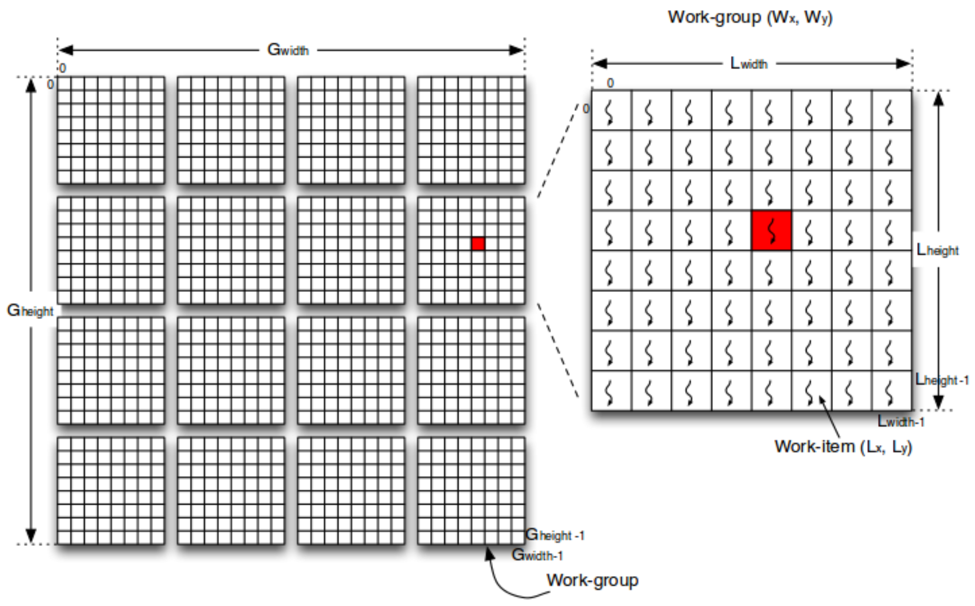


FIGURE 2.3: ID [11]

On a CPU, operating systems often swap two threads on and off execution channels. Threads (cores) are generally heavyweight entities and those context switches are therefore expensive. By comparison, threads on a GPU (work-items) are extremely lightweight entities. Furthermore in GPUs, registers are allocated to active threads only. Once threads are complete, its resources are de - allocated. Thus no swapping of registers and state occurs between GPU threads. [11]

As can be deduced from this section in the underlying memory model, OpenCL is a fairly low-level API. Moreover the programming language used is a derivate of the C language based on C99. A language web developers will most likely be unfamiliar with. Khronos anticipated this and developed the web computing language (WebCL).

2.1.3 WebCL

WebGL and WebCL are JavaScript APIs over OpenGL and OpenCL's API. This allows web developers to create application in an environment they are used to.

In the first place, OpenCL was developed because of web browser's increasing need for more computational power. A necessity which arose from heavy 3D graphics applications such as on-line games and augmented reality. However, OpenCL doesnt provide any rendering capability, it only processes huge amounts of data. That is why OpenCL was designed for inter-operation with OpenGL. WebCL/WebGL interoperability builds on that available for OpenCL/OpenGL. WebCL provides an API for safely sharing buffers with OpenCL. This buffer is inside the GPU which avoids the back and forth copy of data when switching between OpenGL and OpenCL processes. Further precision about the interoperability are discussed in this paper: [12].

GPU computing is quite a new notion. But it is a fast evolving field of research. Single GPUs are not enough anymore, the trend is moving towards GPU clusters.

2.1.4 GPU clusters

Most OpenCL applications can utilize only devices of the hosting computer. In order to run an application on a cluster, the program needs to be split to take advantage of all devices. Virtual OpenCL (VirtualCL) is a wrapper for OpenCL. It provides a platform where all the cluster devices are seen as if located on the same hosting node. Basically, the user starts the application on the master node then VirtualCL transparently runs the kernels of the application on the worker nodes. Applications written with VirtualCL don't only benefit from the reduced programming complexity of a single computer, but also from the availability of shared memory and lower granularity parallelism. Mosix white paper : [13] explains more in depth the OpenCL's functionement.

This first section dedicated to OpenCL introduced how web browsers intend to improve their computational capacities. The second section is on WebSockets. WebSockets is one of the major improvement of brought by HTML5. It provides a revolutionary communication way between client and server. It is also probably the most resource consuming protocol of HTML5 and thus one of the reasons OpenCL and WebCL were developed.

LINK

2.2 Scalability

The growth of distributed computing has changed the way web application are designed and implemented. If compared with today standards, applications used to be deployed so as to say at prototype stage. That is, they were designed to work on a fixed number of servers and not able to ajust as the userbase grows. As the number of connections increases, the load on the servers rises and thus the latency grows. Ideally, an application should aim at a stabil latency, otherwise the application can missbehave.
On the server side, when the nodes begin to be overloaded and struggle to service the client with reasonable response time.
Also, if the servers are overwhelmed they buffer the responses to the clients and then catch up later on . As a result, the clients can be flooded when the load goes down. The sudden rush of message can provoke an unexpected behavior from the servers and can even lead to disconnections.

Nowadays, designing an application without scalability and load balancing in mind is unimaginable. Historically, the reaction to an overloaded the server has always been to scale up.

2.2.1 Scaling up

Scaling up or vertically basically means upgrading the infrastructure. Depending on the needs of the application, the processor, the memory ,the storage or the network connectivity can be improved.

Further performance can be gained by dividing tasks. It only requires to identify the services running idenpendantly or the using message based communication. Those could then be relocated on different nodes.

The main advantage of scaling vertically is it does not involve any software changes and little infrastructure changes. Therefore it is an easy way to increase performances. However for large application, scaling up might prove impossible or at least not economicaly profitable. In case the application already takes advantage of the lastest generation of hardware, the tiniest increase in performance will impact greatly the prices. For example, a high range processor offering ten pourcent more computation power is going to be many times more expensive. Similarly, a memory upgrade could require remplacing all current modules for higher density ones [1].

Moreover, scaling up neither answers availability nor uptime concerns. The system is monolithic and has a single point of failure. Therefore contemporary also scale out to rely on parallel computing.

2.2.2 Scaling out

Scaling out or horizontally, answers most of the problems unsolved by scaling vertically. In a first approach lets ignore the software complexity. Scaling out offers almost unlimitted performance increase and at low cost! If the application is designed to be spread out on multiple nodes, the performance of an infrastructure can be doubled by simply using twice as much servers. Also it is fairly easy to add some redundant server to insure uptime. Plus, compared to scaling up, once the sofware is developped the costs are linear.

When scaling out, the infrastructure implementation is not as much a problem as the code implementation. The expenses are shifted from hardware to developpment costs.

Code implementation

Developping a parallel code is quite complicated and all applications can not be parallized. In 1967 Gene M. Amdhal defined the so called Amdahl's law which is still used today to define the maximum to expect when parallelizing a code [14]. Each software can be divided in two separete parts, the parallel part and the sequential part. Parallel computing does not improve the sequential part. If a the code is mainly sequential, then increasing the number of processors will only cause the parallel part to finish first and stay idle waiting for the sequential part to finish.
Assuming P is the portion of a program that can be parallelized and 1 - P is the portion that remains serial, then the maximum speedup that can be achieved using N processors is:

```
Speedup = 1/ [ (1-P) +P/N ]
```

If 70% of the program can be run in parallel (P = 0.7) the maximum expected speedup with 4 processors would be:

```
  Speedup = 1/[ (1-0.7) + 0.7/4 ] = 2.105
```

When the N tends to infinity, P/N tends to zero. As a result when the number of processors reach a certain point, the speed up will be 1/(1 -P).

```
Speedup = 1/ (1-0.7) = 3.33
```

CHANGE NUMBER add N oo

Nathan T. Hayes's paper for Sunfish Studio [15] studies the how parallel computing can profit the motion picture industrie. The following chart present the maximum speedup which can be expected from an application in function of the pourcentage of parallel code in the programme.

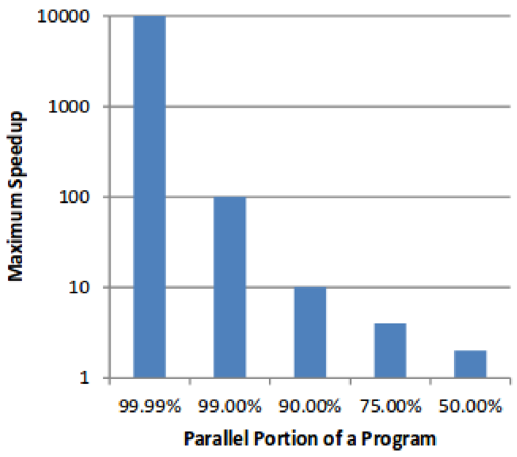


FIGURE 2.4: Amdahl law [15]

However, Amdahl's law is based on assumption which are hardly verified in pratique. Following are sumed up reasons not to give to much importance to Amdahl's law [16]

- The number of thread is not always always equivalent to the number of processors.
- The parallel portion does not have a perfect speedup. Computation power is used for communication between processus. Also some ressources like caches and bandwidth have to be shared across all the processors.
- Allocating, delocating and switching threads introduce overhead, overhead growing linearly with the number of thread.
- Even an optimized code will not have perfectly synchronised threads, at some point some processus will have to wait for others to finish.

Amdahl's law has long been used as an argument against massively parallel processing. In 1988 Gustafson law came as an alternative to Amdahl's law to estimate the speedup. In both law, the sequential portion of the problem is supposed to stay constant. But in Gustafson's law the overall problem size grows proportionally to the number of cores. As a result, Gustafson's gives slightly different results to Amdahl's and encourage the use of parallel computing.

However later studies tends to contest the legimity of both laws. Yuan Shi's paper [17] even proves both theory are but two different interpretations of the same law. He concludes his study by saying these laws are too minimalist and what computer scientist really need is a practical engineering tool that can help the community to identify performance critical factors.

Infrastructure implementation

Figures x, shows a web application where each independent services have already been splitted on different servers. Assuming the hardware is also optimized, this application cannot be increased further by scaling up techniques. The only way to increase performance is to scale out and use for example three servers for the web services. However this introduces a new problem, a third party must command these servers. This server also called load balancer is in charge of distributing the work between the worker and completely hides the complexity to the user.

2.2.3 Load balancing

Once the system is spread across a cluster, the final step to distribute the application is to plan how requests are dispatched across the available ressources.

Chapter 3

Experiment

3.1 Client side

3.1.1 Client server

SocketCluster-client makes the instanciation of a WebSocket clients on one core quite straightforward. To deploy it on all available nodes, node.js `fork()` function is used. A client code example is given in appendix A.

The first experiment is a safety test. It checks if `fork()` distributes evenly the work among the cores.

Parameters	
Instance type	amazon s3 m3.2xlarge
Experiment time	120 s
Number of new communication created at each iteration	15
Client creation period	1 s
Type of ping	random number
Ping period	2.5 s

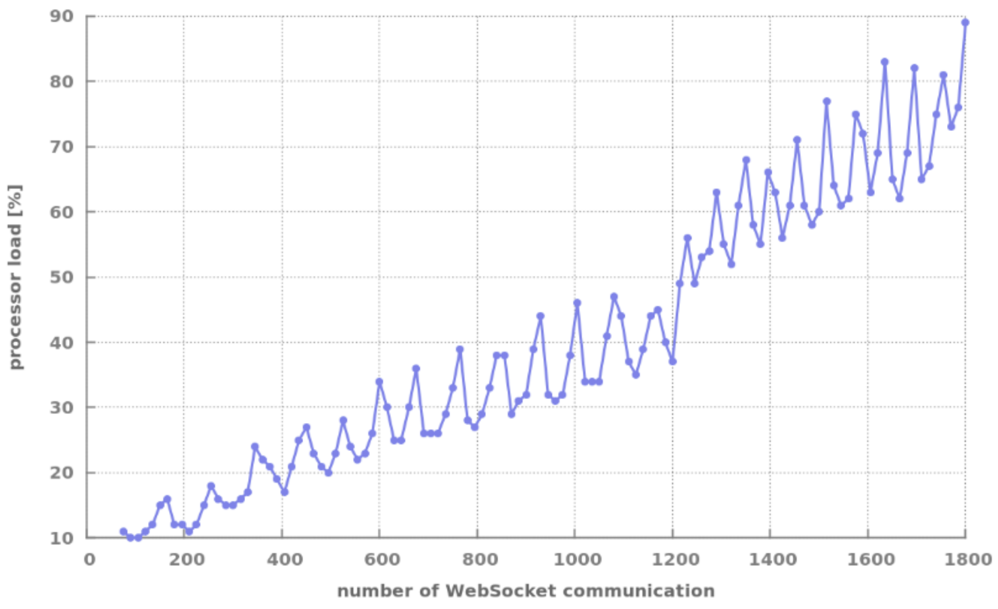


FIGURE 3.1: One client throughout

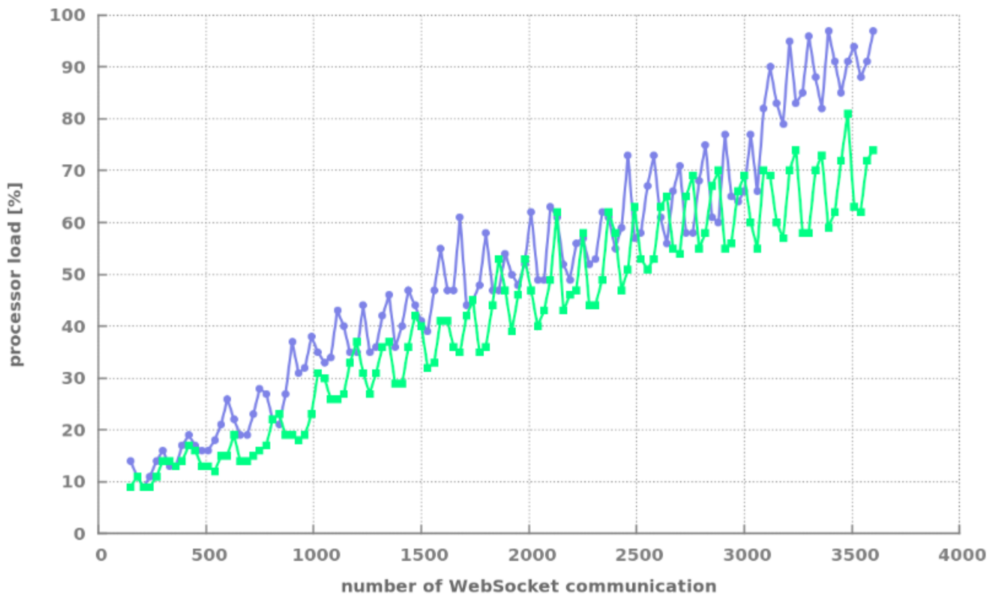


FIGURE 3.2: Two clients throughout

From 3.1 and 3.2 can be inferred that the client implementation works flawlessly. Adding a second core enables twice as much communication to be established.

3.1.2 Client browser

Browsers can be configured to connect to one of the SocketCluster worker. For example if the experiment is run locally, typing `localhost:8080` in the url will make the browser listen to the port 8080. After what sending ping thanks to `socket.emit('ping')` can help to check the throughout performance. It also displays the growing number of pings a working has sent since begenning of the experiment.

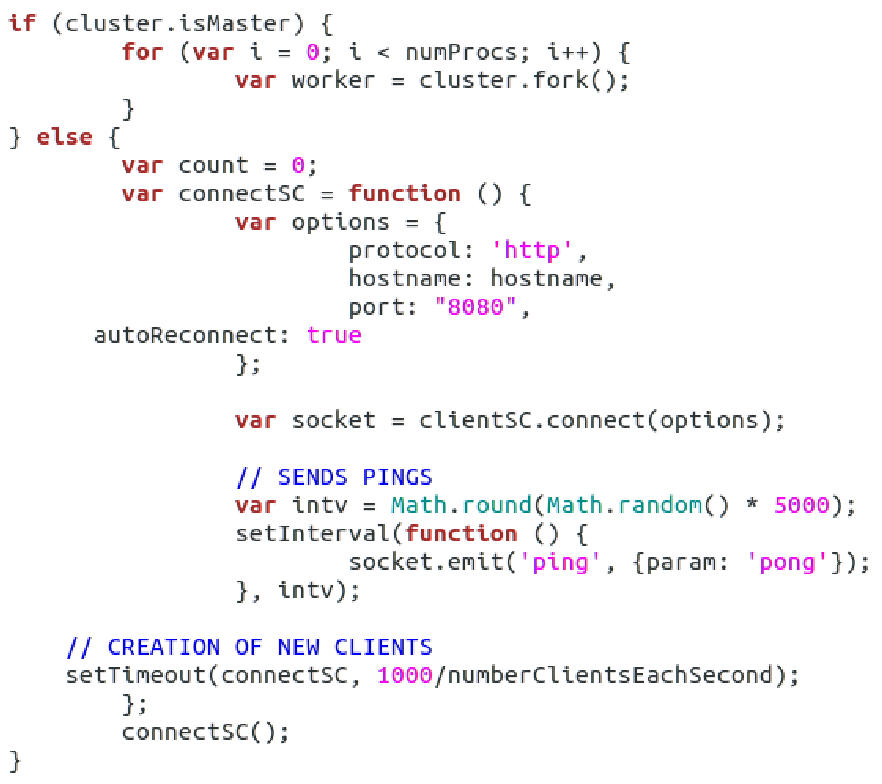
INSERT GRAPH

3.2 Server side

Appendix A

Client side code

This is an example of a WebSocket client code spread on all available cores. New clients are spawned every `numberClientsEachSecond`. Thereafter, every `intv` each clients sends a ping event cast to a Javascript JSON object.



```
if (cluster.isMaster) {
        for (var i = 0; i < numProcs; i++) {
                var worker = cluster.fork();
        }
} else {
        var count = 0;
        var connectSC = function () {
                var options = {
                        protocol: 'http',
                        hostname: hostname,
                        port: "8080",
      autoReconnect: true
                };

                var socket = clientSC.connect(options);

                // SENDS PINGS
                var intv = Math.round(Math.random() * 5000);
                setInterval(function () {
                        socket.emit('ping', {param: 'pong'});
                }, intv);

    // CREATION OF NEW CLIENTS
    setTimeout(connectSC, 1000/numberClientsEachSecond);
        };
        connectSC();
}
```

FIGURE A.1: Client code

Bibliography

[1] J. Mogul R. Fielding and J. Gettys. Hypertext transfer protocol – http/1.1. *Request for Comments 2616*, 1.4 Overall operation, September 2004. URL http://www.w3.org/Protocols/rfc2616/rfc2616-sec1.html#sec1.

[2] Eliot Estep. Mobile html5: Efficiency and performance of websockets and server-sent events. *Master thesis*, 3.3 Web techniques, June 2013. URL http://nordsecmob.aalto.fi/en/publications/theses2013/thesis_estep/.

[3] Peter Lubbers and Frank Greco. Html5 web sockets: A quantum leap in scalability for the web. March 2010. URL http://www.websocket.org/quantum.html.

[4] I. Fette and A. Melkinov. The websocket protocol. *Request for Comments 6455*, December 2011 2011. URL http://tools.ietf.org/html/rfc6455.

[5] Tobias Oberstein. Dissecting websocket's overhead. *Tavendo*, January 2014. URL http://tavendo.com/blog/post/dissecting-websocket-overhead/.

[6] Tomislav Capan. Why the hell would i use node.js. *toptal*, February 2013. URL http://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js.

[7] Mikito Takada. Understanding the node.js event loop. *Mixu's tech*, February 2011. URL http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/.

[8] Deniz Ozger. Finding the right node.js websocket implementation. *medium*, January 2014. URL https://medium.com/@denizozger/finding-the-right-node-js-websocket-implementation-b63bfca0539.

[9] David Gohara John E. Stone and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12:66–73, June 2010. URL http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=5457293&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D5457293.

[10] Tomi Aarnio Janne Pietiinen Jari Nikara Eero Aho, Kimmo Kuusilinna. Towards real-time applications in mobile web browsers. *Embedded Systems for Real-time Multimedia*, pages 57–66, October 2012. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6507030.

[11] Mikal Bourges-Svenier. Graphics programming on the web, webcl course note. *Special Interest Group on GRAPHics and Interactive Technique conference*, October 2012. URL http://khronosgroup.github.io/siggraph2012course/WebCL/WebCL%20Course%20Notes.pdf.

[12] Tasneem Brutch Won Jeon and Simon Gibbs. Webcl for hardware-accelerated web applications. *Tizen developer conference*, May 2012. URL http://dev.bowdenweb.com/html/api/webcl/webcl-for-hardware-accelerated-web-applications-dev-track-008.pdf.

[13] A. Barak and A. Shiloh. The virtualcl (vcl) cluster platform. *Mosix white paper*, 2012. URL http://www.mosix.cs.huji.ac.il/vcl/VCL_wp.pdf.

[14] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, pages 483–485, 1967. URL http://www.futurechips.org/wp-content/uploads/2011/06/5_amdahl.pdf.

[15] Nathan T Hayes. High performance parallel computing in the motion picture industry. *Sunfish white paper*, February 2012. URL http://sunfishstudio.com/downloads/MeridianWhitePaper.pdf.

[16] Aaeter Suleman. Parallel programming: When amdahl's law is inapplicable. *Future chips*, June 2011. URL http://www.futurechips.org/thoughts-for-researchers/parallel-programming-gene-amdahl-said.html.

[17] Yuan Shi. Reevaluating amdahl's law and gustafson's law. October 1996. URL http://spartan.cis.temple.edu/shi/public_html/docs/amdahl/amdahl.html.