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

© Cranfield University, 2014. All rights reserved. No part of this publication may be reproduced without the written permission of the copyright owner.

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

Declaration of Authorship i

Abstract ii

Acknowledgements iii

List of Figures vi

Abbreviations vii

1 Introduction 1
- 1.1 Javascript . . . 1
- 1.2 Client server communications . . . 1
  - 1.2.1 HTTP protocol . . . 2
  - 1.2.2 Page by page model . . . 2
  - 1.2.3 Polling . . . 3
  - 1.2.4 Long polling . . . 4
  - 1.2.5 Streaming . . . 6
  - 1.2.6 Current technologies in browser . . . 6
- 1.3 WebSocket protocol . . . 7
  - 1.3.1 Definition . . . 7
  - 1.3.2 The WebSocket handshake . . . 8
  - 1.3.3 Transport layer protocol . . . 9
  - 1.3.4 The WebSocket frame anatomy . . . 10
  - 1.3.5 Proxies . . . 11

2 Literrature review 13
- 2.1 Implementation . . . 13
  - 2.1.1 WebSocket server implementation . . . 13
  - 2.1.2 Heterogeneous implementation with OpenCL . . . 15
  - 2.1.3 WebCL . . . 19
  - 2.1.4 GPU clusters . . . 20
- 2.2 Scalability . . . 21
  - 2.2.1 Scaling up . . . 21

2.2.2 Scaling out 22

3 Design and Implementation **26**

4 Experiment **28**
4.1 Client throughout 28
4.1.1 Client scalability 28
4.1.2 browser testing 30
4.2 Comparaison with engine.io 30
4.3 SocketCluster context switching 33
4.4 35

A SocketCluster **38**
A.1 Simple ping-pong exchange 38
A.2 File transfer 39

B Engine.io **41**

C Real time throughout check **43**

Bibliography **44**

List of Figures

1.1 Client server communication 3
1.2 Polling 4
1.3 Long polling 5
1.4 Streaming 6
1.5 Frame overhead 10
1.6 TLS overhead 10
1.7 TCP overhead 10
1.8 Websocket messages sent individualy 11
1.9 Batched WebSocket messages 11

2.1 Plateform model 17
2.2 Memory model 18
2.3 Work - group 19
2.4 Amdahl law 24

4.1 Client throughout 29
4.2 Browser connection to SocketCluster 30
4.3 WebSocket implementation 31
4.4 Engine.io implementation 32
4.5 Contexting switching 34
4.6 Simple WebSocket client 35
4.7 WebSocket server on three cores 36
4.8 WebSocket server on five cores 36
4.9 WebSocket server on seven cores 37

A.1 Simple WebSocket client code 38
A.2 Simple WebSocket server code 39
A.3 Client code for file transfers with WebSocket 40
A.4 Server code for file transfers with Websocket 40

B.1 Engine.io client code 41
B.2 Engine.io server code 42

C.1 Modification to index.html 43

Abbreviations

RFC	Request For Comment
HTTP	HyperText Transfert Protocol
HTML	HyperText Markup Language
TCP	Transmission control protocol
IP	Internet Protocol
UDP	User Datagram Protocol
OpenCL	Open Computing Language
OpenGL	Open Graphic Library
API	Application Programming Interface
GPU	Graphic Processing Unit
GPGPU	General Purpose computation on GPU
CPU	Computing Processor Unit
SIMD	Single Instruction Multiple Data
MIME	Multi purpose Internet Mail Extension
DSP	Digital Signal Processing
VCL	Virtual open Computing Language
RFC	Request For Comment
RFC	Request For Comment
RFC	Request For Comment
RFC	Request For Comment
RFC	Request For Comment
RFC	Request For Comment
RFC	Request For Comment
RFC	Request For Comment

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.

CHANGE !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

1.2 Client server communications

This section studies the evolution of client server communication, beginning with the page by page model until current technologies. However as an introduction, the first part is about HTPP which is the fundation of client server communication.

1.2.1 HTTP protocol

The HTTP protocol is a request/response protocol defined in the request for comment (RFC) [1] as follows:

```
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 requestmodifiers, 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.
```

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 nineties, most web pages were static. As a consequence, the communication between client and server were rather limited. Typically, the client would send occasional request to the server. The server would then answer but all communication would stop there until a new event was triggered by the user.

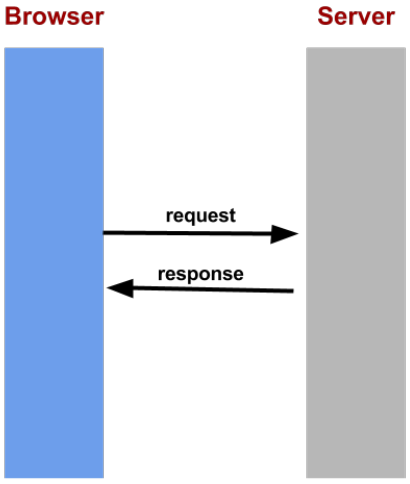

FIGURE 1.1: Client server communication

The notion of dynamic web appeared in 2005 with the introduction 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 Polling

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

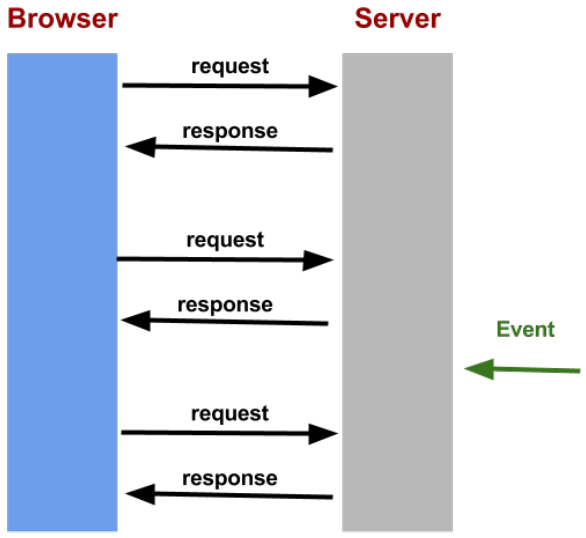

FIGURE 1.2: polling

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].

1.2.4 Long polling

Long polling is based on Comet technologies and is a slight step further toward server sent 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.

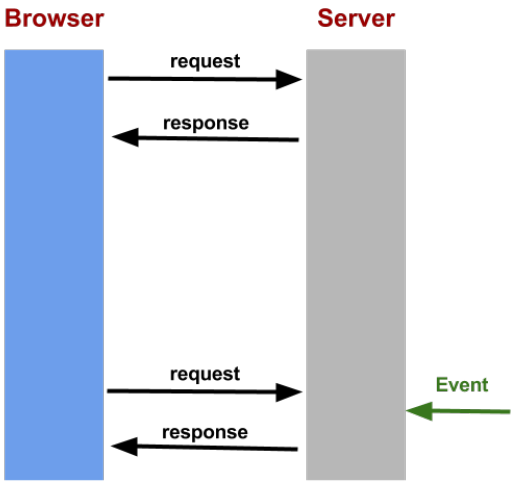

FIGURE 1.3: Long polling

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, request 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 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.

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].

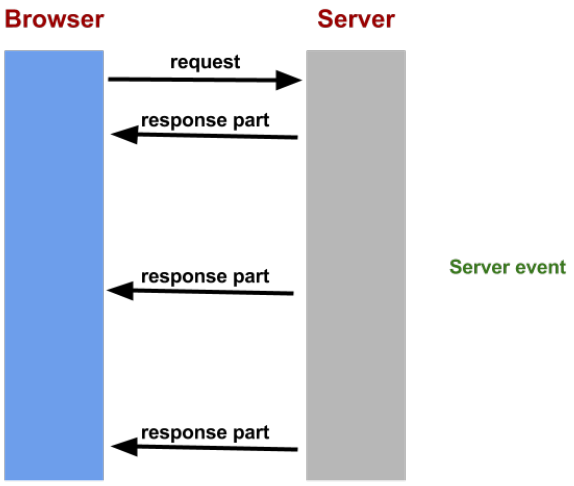

FIGURE 1.4: Streaming

Wouldn't it be because of proxies, streaming would be 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].

1.2.6 Current technologies in browser

At the moment, comet technologies are still the most popular way of communication between browsers and servers. Techniques 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.

This section is a study of the WebSocket protocol. To begin with it defines the protocol. Secondly it studies how to establish a WebSocket connection. Afterwards it goes on with an in depth study of WebSockets' transport layer and frame anatomies. And to finish it provides a brief discussion of WebSockets' interaction with proxies.

1.3.1 Definition

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

```
The WebSocket Protocol enables two-way communication between a
clientrunning 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.
```

To iniate a WebSocket communication, first a HTTP handshake needs to be done.

1.3.2 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.

1.3.3 Transport layer protocol

The internet is based on two transport layer protocols, 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 ensure the consistancy of the data. TCP is said to be a stream oriented because the data is sent in independent segments.

UDP

UDP is unreliable but fast. The protocol offers no guarenty 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 example of WebSockets' use cases. They require low latency and have a high rate of update. To achieve 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 relies on a few other protocols. Namely HTTP to initialize the communication , ethernet, TCP/IP and finaly TLS in case a secure connections is required. The next subsections studies the influence this protocols have in the anatomy of WebSockets frame.

1.3.4 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 **??**, 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 protocols of other layers required for WebSocket's functionement. Figure **??** and **??** respectively show 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 thirteen bytes. In comparaison 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 all the protocols revolving around WebSockets don't dwarf the overhead of the WebSocket protocol 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 **??**. The individual messages are put in a queue and batched in a single Websocket message like in figure 1.9.

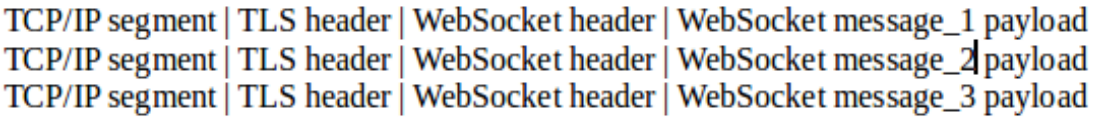

FIGURE 1.8: WebSocket messages sent individualy [5]

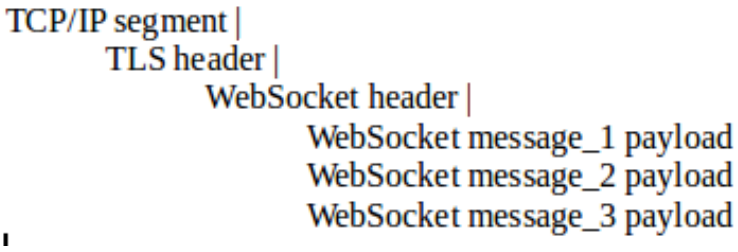

FIGURE 1.9: Batched WebSocket messages [5]

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

1.3.5 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

Chapter 2

Literrature review

This second chapter studies research done around WebSockets. The first section is about the implementation of WebSockets server, the second is about scalability.

2.1 Implementation

As in any project, in order to avoid future technical problems, it is better to first study similar projects. The goal of this implementation study is to find a suitable language and possibly a good library to run the experiment.

2.1.1 WebSocket server implementation

In order to narrow the library study, first a language needs to be selected.

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. Node.js seems to be the perfect environment for

this study, therefore other languages will deliberatly be left apart and the focus will be on explaining why Node.js is appropriate.

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's 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 it tries to connect to a server via WebSocket, in case it fails 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.

In conclusion, Node.js and its real-time library engine.io seems a good choice for our experiment. However better perfomance could be reached using an heterogeneous implementation.

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 not only allows 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 [5].

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 claim 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 to be managemed by a host CPU. Fortunately OpenCL is designed to be used in heterogeneous environment. It abstracts CPUs and GPUs as compute devices. This way, 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 hardest 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 in the following paper: [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 SIMD threads or processing elements with its own memory. [11]

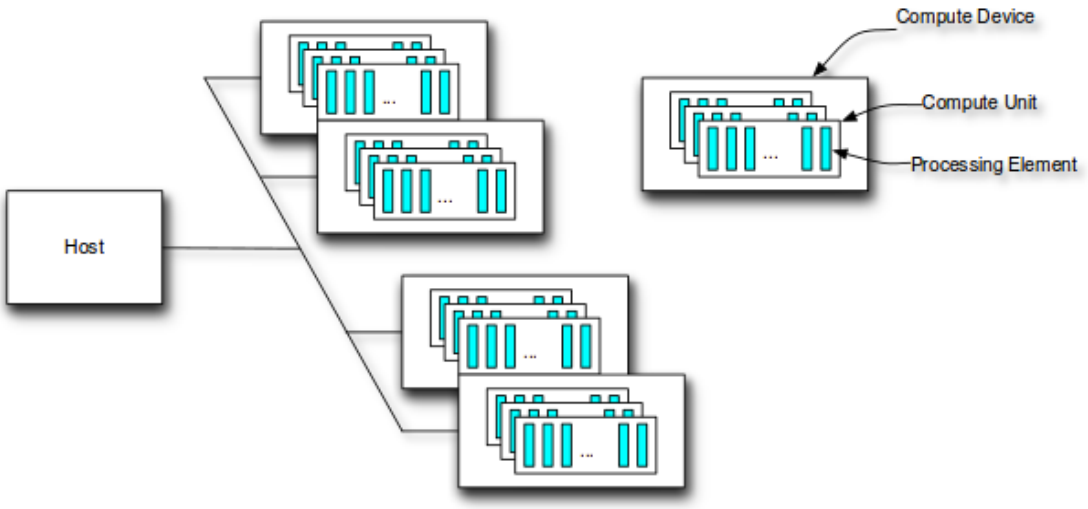

FIGURE 2.1: Plateform model [11]

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 [11].

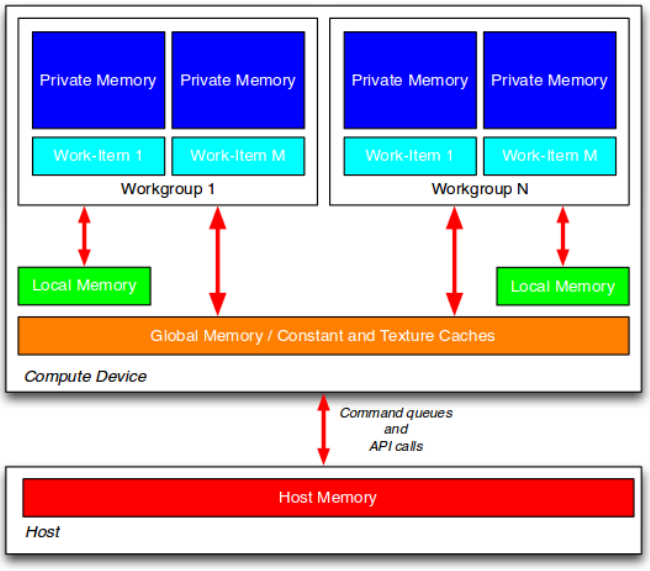

FIGURE 2.2: Memory model [11]

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 workgroups. This is the smallest unit of parallelism on a device. Individual work-items in a workgroup 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].

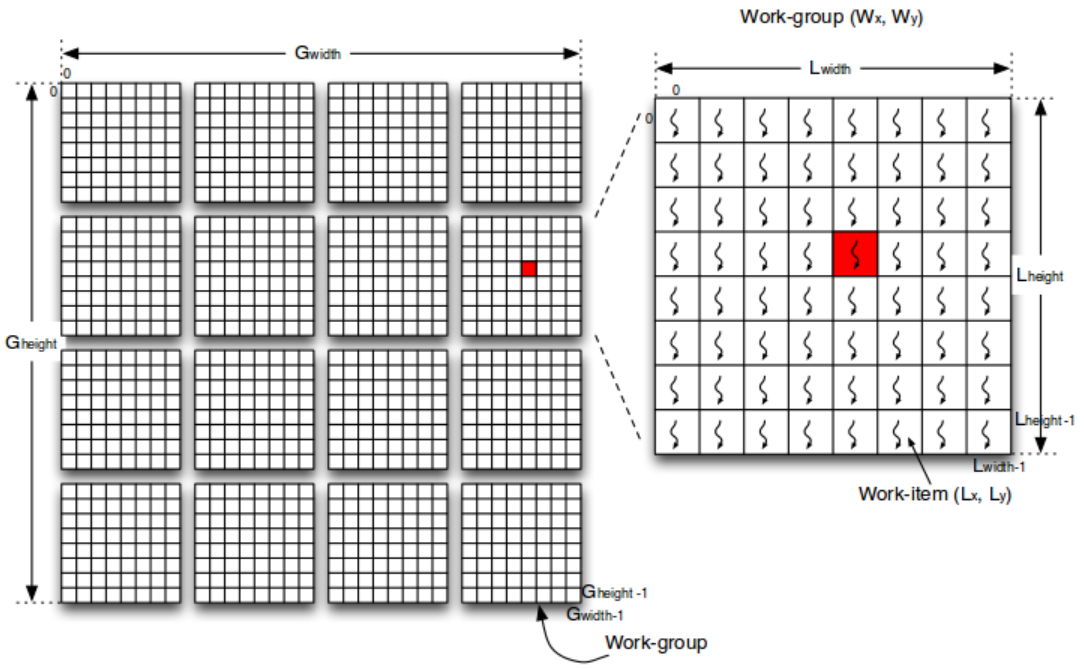

FIGURE 2.3: Work Group [11]

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 or state occurs between GPU threads. [11]

As can be deduced from this section about the underlying memory model, OpenCL is a fairly low-level API. In fact, 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 VCL's functionement.

OpenCL and VirtualCL are powerful tool to create highly parallel clusters. But current implementation with CPUs only already reach a million concurrent connections [**?**]. So far there is simply no need for more powerful clusters.

However, all company don't have acccess to dual Quad-core Xeon CPUs used in Kaazing cluster to reach a million concurrent connections. Usual practice is to build a scalable cluster, to ajust computing power in fonction of the needs.

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, the nodes will 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 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 idependantly 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 infrastrucure is already equiped with the lastest hardware generation, the tiniest increase in performance will impact greatly the price. 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.

Moreover, scaling up neither answers availability nor uptime concerns. The system is monolithic and has a single point of failure. Therefore contemporary project usually scale out and use 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:

$$speedup = \frac{1}{(1-P) + \frac{P}{N}}$$

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

$$speedup = \frac{1}{(1-0.7) + \frac{0.7}{N}} = 2.1$$

When the number of processors reaches a certain point, the speed up will be:

$$\lim_{n\to\infty} speedup = \frac{1}{1-P} = 3.3$$

Nathan T. Hayes's paper for Sunfish Studio [15] studies 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.

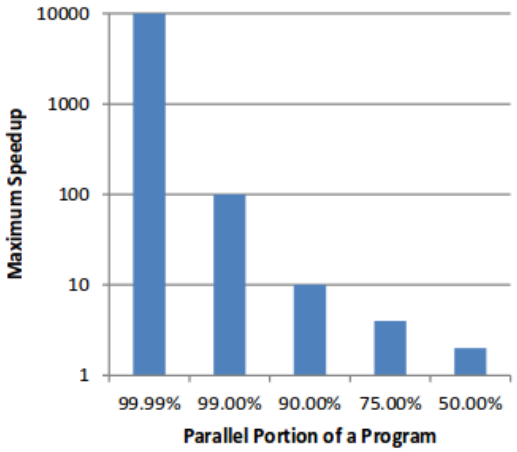

FIGURE 2.4: Amdahl law [15]

The figures speak for itslelf, to envisage parallel computing, the portion of parallel code must be very high.

SPEAK ABOUT EMBARASSING PARALLELISM ?

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

Beside coding complication, scaling out also brings infrastructure changes. A third party must be in command of all servers. This master server is also called load balancer. Its role is to distribute the work evenly between the workers and thus completly hides the complexity to the user.

Chapter 3

Design and Implementation

Current research around WebSocket are centered on distributed computing. Either on CPUs architecture, GPUs architecture or heterogeneous architecture. For the time being, clustering WebSocket servers is rather difficult and reserved to researcher or specialized companies. Actually in Node.js, there is hardly any library to simply and efficiently implement a multi-core server. Node.js single thread nature is a double edge sword. On one side it will allow more concurrent connections to be established but it also means special attention needs to be given to run the code on all the servers cores. SocketCluster is brand new real-time engine aiming exactly at that. At this point of my thesis I had to make a choice between either the theoretical study of WebSocket clusters or the benchmarking of SocketCluster. After contacting Jonathan Gros-Dubois, the creator of SocketCluster, I made up my mind for the latter. Indeed, SocketCluster being under development the tests carried out so far are rather sparse.

SocketCluster

As described on the github project [**?**], SocketCluster is a fast, highly scalable HTTP and WebSocket server. It facilitates the creation of multi-process realtime application that make use of all CPU cores on a machine/instance. Therefore removing the limitations of having to run a Node.js server as a single thread. SocketCluster's focus is on vertical scaling. If N is the number of cores available on the server, then SocketCluster is N time than any comparable single-threaded WebSocket server. Under the hood, the application deploys itself on all available cores as a cluster of process. The process can be grouped in three categories:

stores, workers and load balancers. The experiments carried out in the next chapter aim at establishing a ratio rule depending on the number of clients connected and the type of messages they send.

U-limit

My first experiments with WebSockets were far from satisfactory. Past at total 512 communication, new sockets were inexplicably crashing. This comes from a system limit set up on linux operating systems. By default the maximum number of file that can be sent over tcp is 1024.

Fortunately, this limit can easily be increased by appending this line:

C 10K Problem

The C 10K is a challenge issued in 1999 by Dan Kegel it consist in reaching 10 000 concurent client connections. Engineers solved this problem by fixing operating systems kernel and creating new single threaded programming languages like Node.js. Today's objective is rather to achieve 10 000 000 concurrent connections like mentioned in the exellent article in highscalability.com [**?**]. Such amount of connections is beyond the scope of this thesis, but apparently the solution to improve the number of connections is to move heavy lifting from the kernel to the application itself.

Monitoring tool

Generally monitoring tools are recording the average CPU usage. Some can be configured to record processor usage on each cores. But ideally our experiment would require to record the processor usage for each threads. For this purpose and to have more control over the data, I choose not to use an out of the box tool. I am using top's batch mode to output the processor usage in a file. After what I am doing some bash operation to formate the data properly. And finally I am plotting the graphs with gnuplot.

Chapter 4

Experiment

4.1 Client throughout

4.1.1 Client scalability

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.1.

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

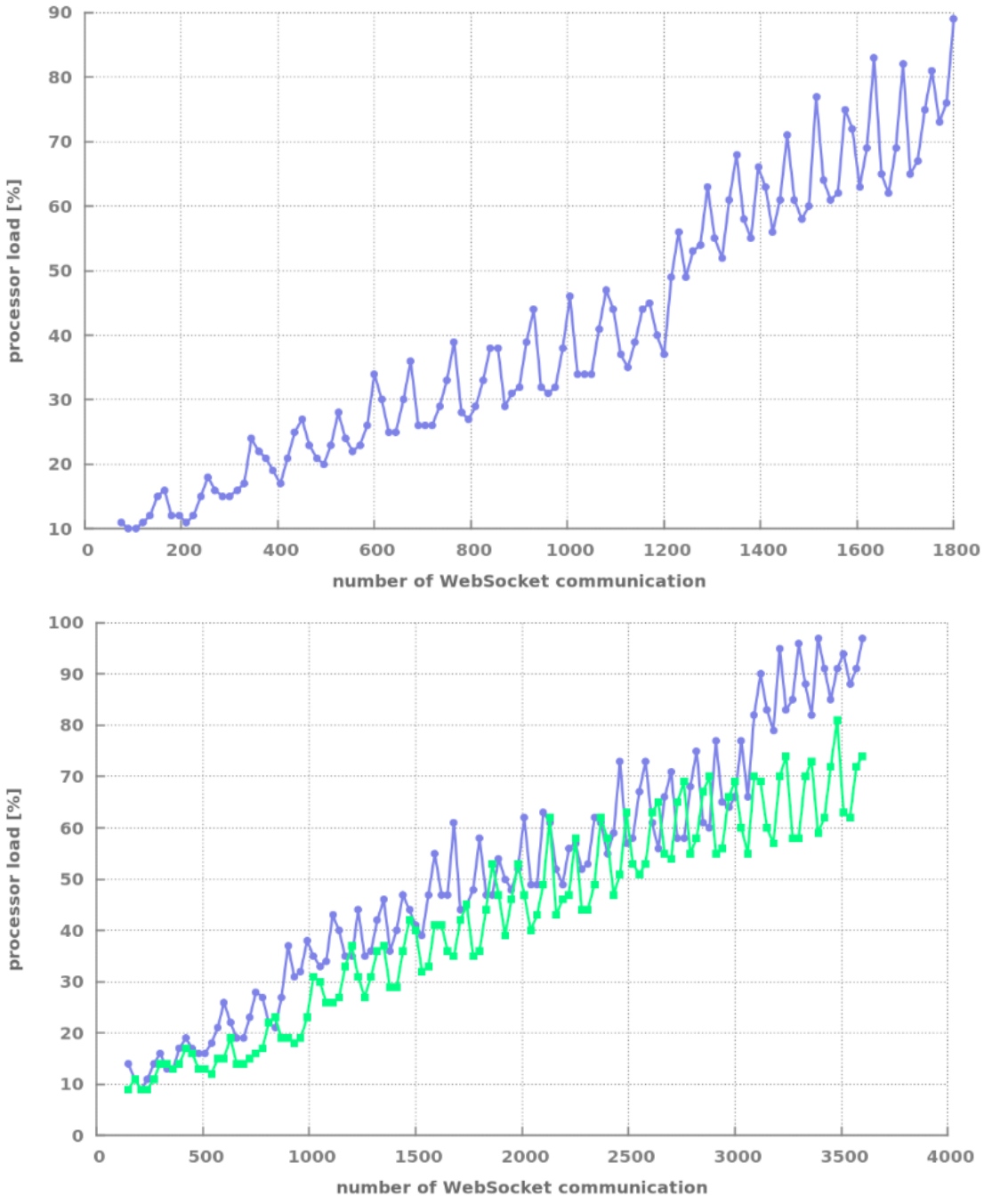

FIGURE 4.1: Client throughout

From figure 4.1 can be inferred that the client implementation works flawlessly. Adding a second core enables twice as much communication to be established.

4.1.2 browser testing

As mentionned in appendix C.1, by opering minor changes in the `index.html` file, the browser can be configured to display in real time the number of pings received by a particular worker. If the experiment is running locally, typing `localhost:8080` in the url will link the browser to one worker.

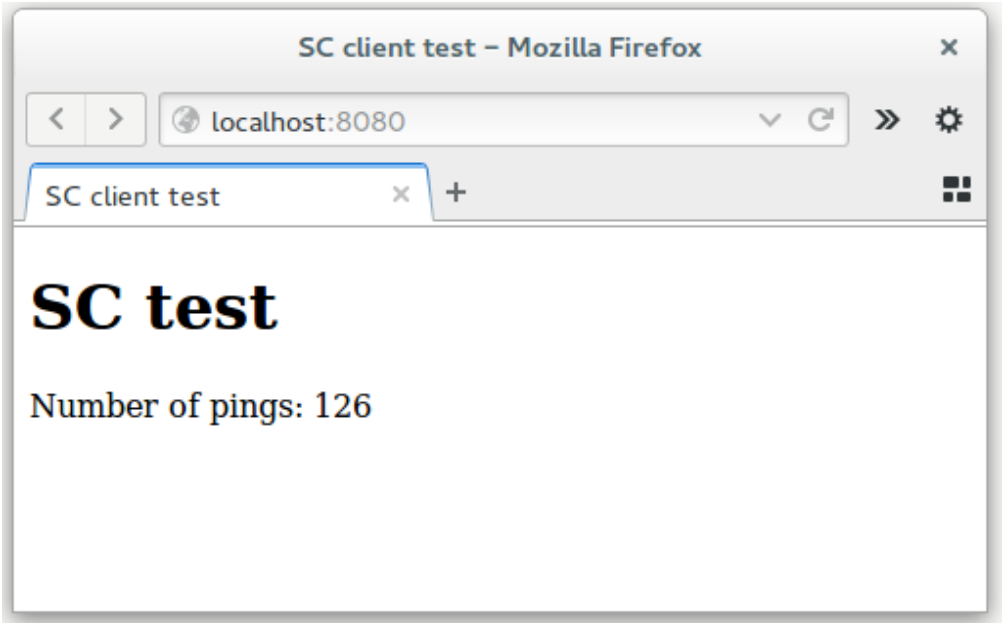

FIGURE 4.2: Browser connection to SocketCluster

By doing so we can embody a user connected to our WebSocket server and have a better idea of the reactivity of the server.

4.2 Comparaison with engine.io

SocketCluster has been created to ease the creation of multi - core WebSocket server. Logicaly the first experiment carried out on the server was to compare a WebSocket to a traditionnal Engine.io server.

Engine.io and SocketCluster codes can be found in appendix A and B.

Parameters	
Instance type	amazon ec2 m3.2xlarge
Experiment time	60 s
Number of new communication created at each iteration	20
Client creation period	1 s
Type of ping	random number
Ping period	2.5 s
Number of clients	2

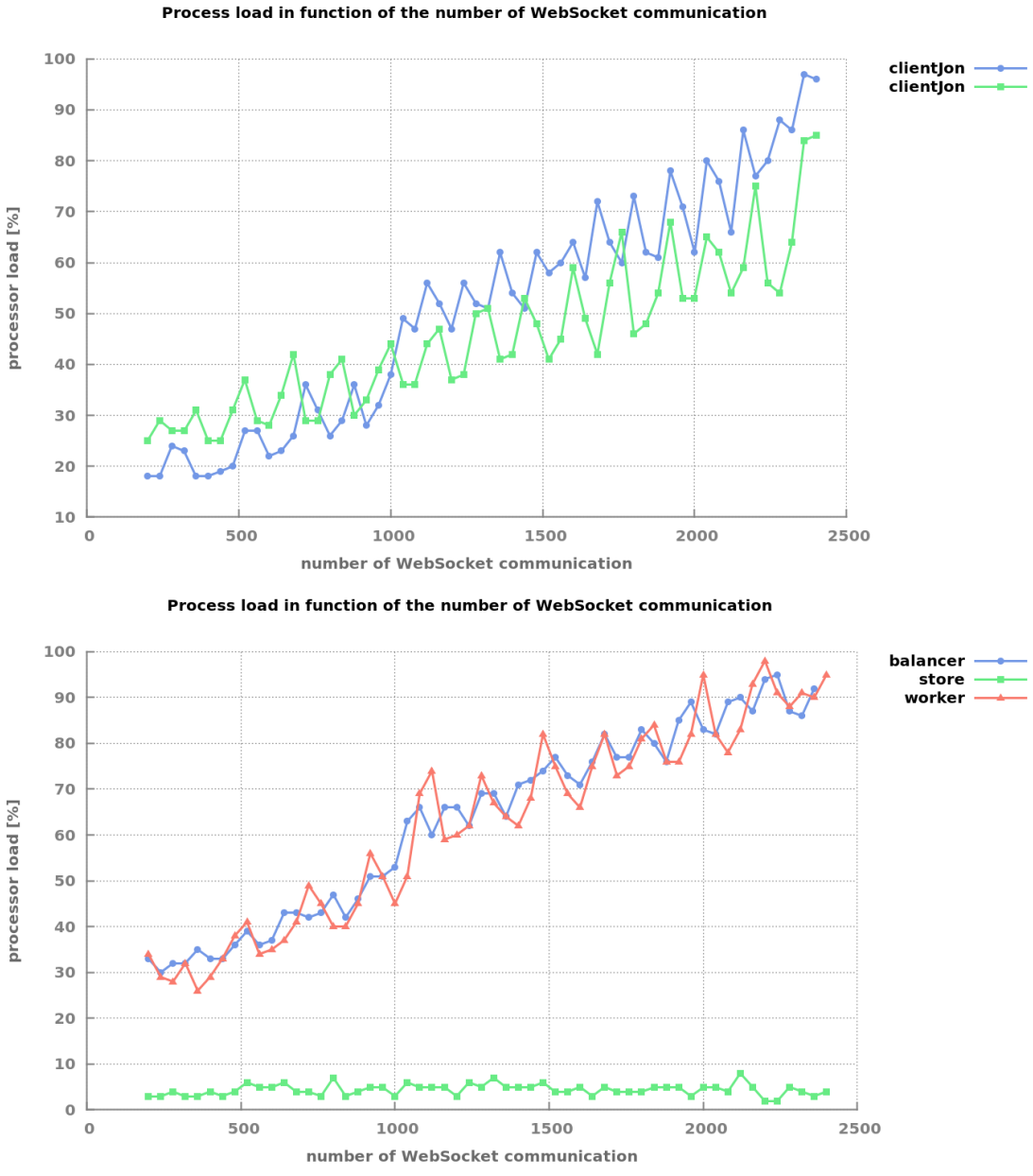

FIGURE 4.3: WebSocket implementation

In this experience, two clients are used to achieve a maximum of 2400 WebSocket communications. The server was configured to use one storage, one load balancer and one worker. While the store processor is quite idle, the two other processors on the other hand are almost used at full capacity.

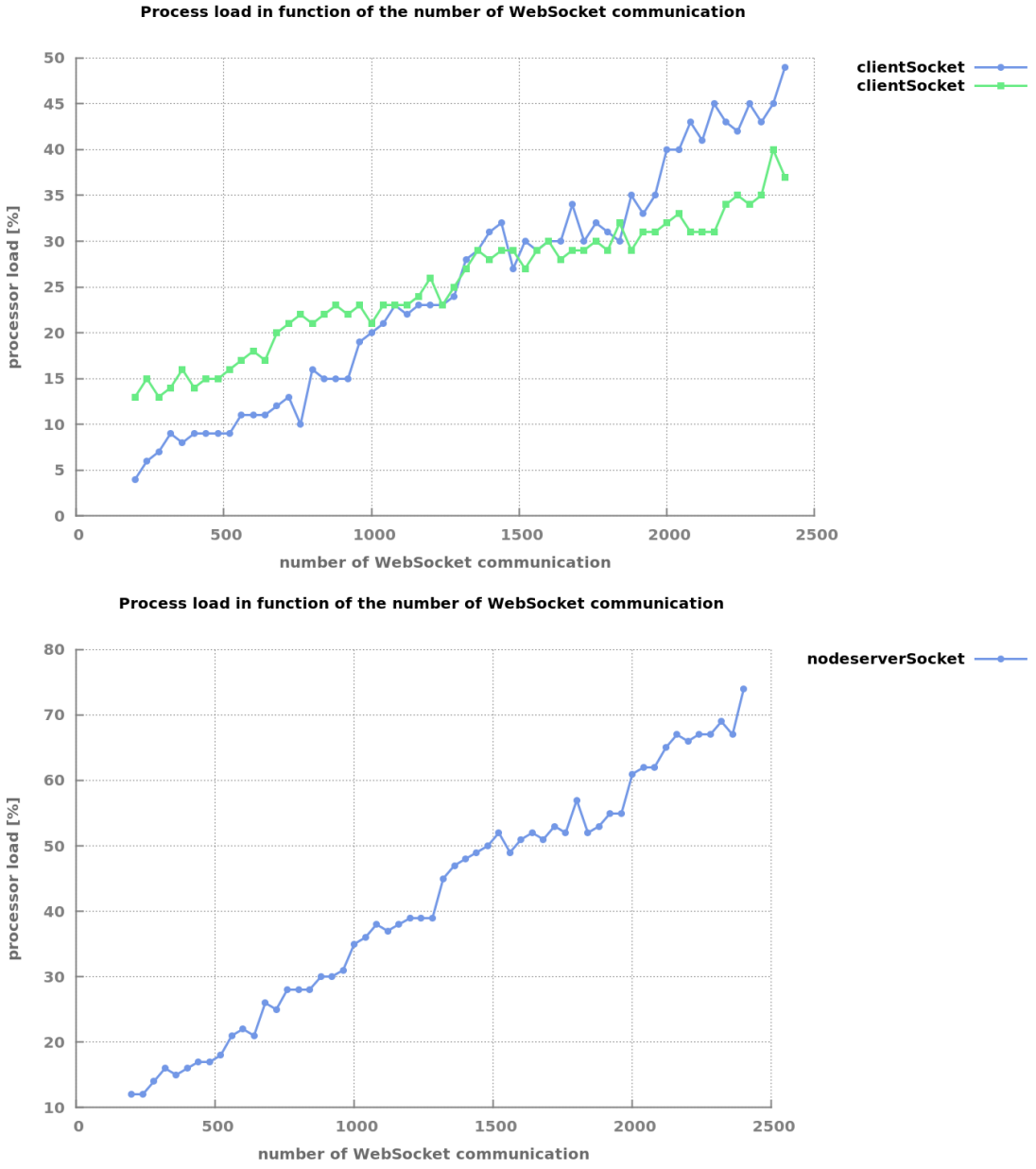

FIGURE 4.4: Engine.io implementation

Surprisingly, pure engine.io implementation seems to be more efficient. Clients are hitting a maximum of 50% processor usage compared to 90% for WebSockets.

On the server side also, engine.io processor peaks at 75% compared to almost 100% for WebSockets. Also even if both code have been deployed on similar virtual

machines: `amazon ec2 m3.2xlarge` the engine.io server is running only only on one core compared to three for SocketCluster (one storage, one load balancer and one worker). This seems to show, SocketCluster is not adapted to low number of communication.

An interesting experiment worth doing at this point, is to try to use SocketCluster on one core.

4.3 SocketCluster context switching

For this experiment a single core virtual machine is used for the server: `amazon ec2 m3.medium`.

Parameters	
Server instance type	amazon ec2 m3.medium
Client instance type	amazon ec2 m3.2xlarge
Experiment time	80 s
Number of new communication created at each iteration	40
Client creation period	1 s
Type of ping	random number
Ping period	2.5 s
Number of clients	2

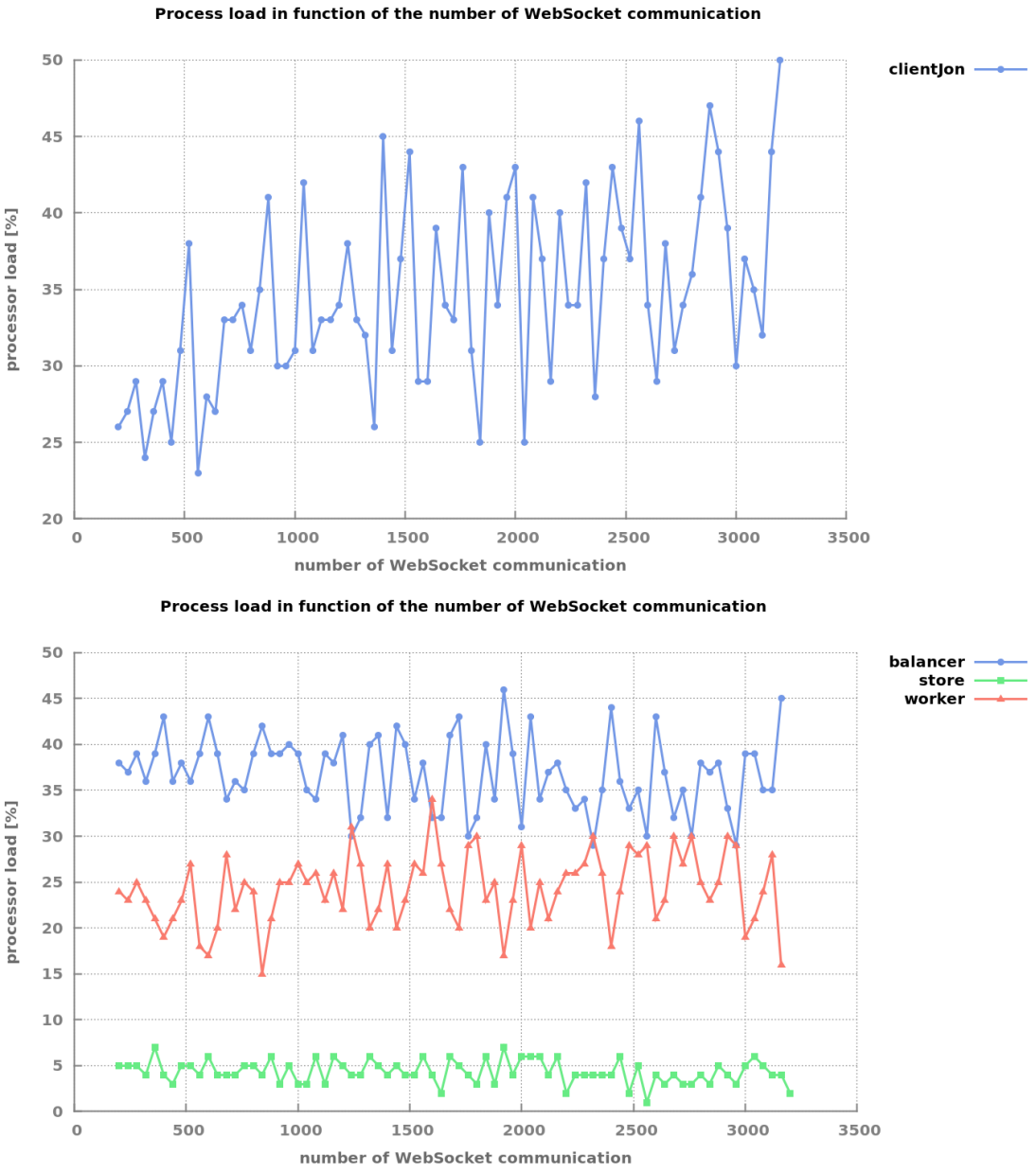

FIGURE 4.5: Context switching

At the first glimpse, anyone can immediately tell there is a problem with the server graph. The Load seems to vary randomly at an average of 40%. What really happens, is that most WebSocket connections are dropped shortly after beeing created or they not even created. The problem is a single core needs to handle four threads. So each time another application is called the context changes. The result is even worse case of a multi-procesor server, because threads then are balanced between processor. Threads are heavy weight units, moving them introduces consequent overheads.

to conclude, this experiment prooves SocketCluster is not aimed to be used with project which imvolve more threads than available cores.

4.4

Client code

The client code used in all this part is the same. Two clients are used to produce a maximum of 2400 WebSocket communications.

Parameters	
Instance type	amazon ec2 m3.2xlarge
Experiment time	60 s
Number of new communication created at each iteration	20
Client creation period	1 s
Type of ping	random number
Ping period	2.5 s
Number of clients	2

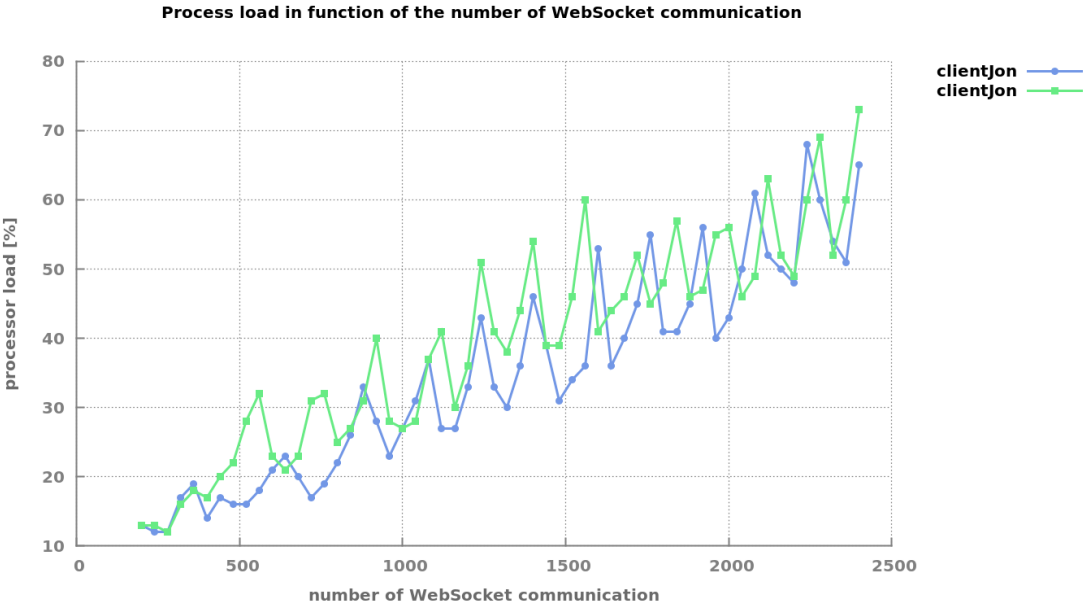

FIGURE 4.6: client code

Server code

The first test is run a server using a one store, one load balancer and one worker.

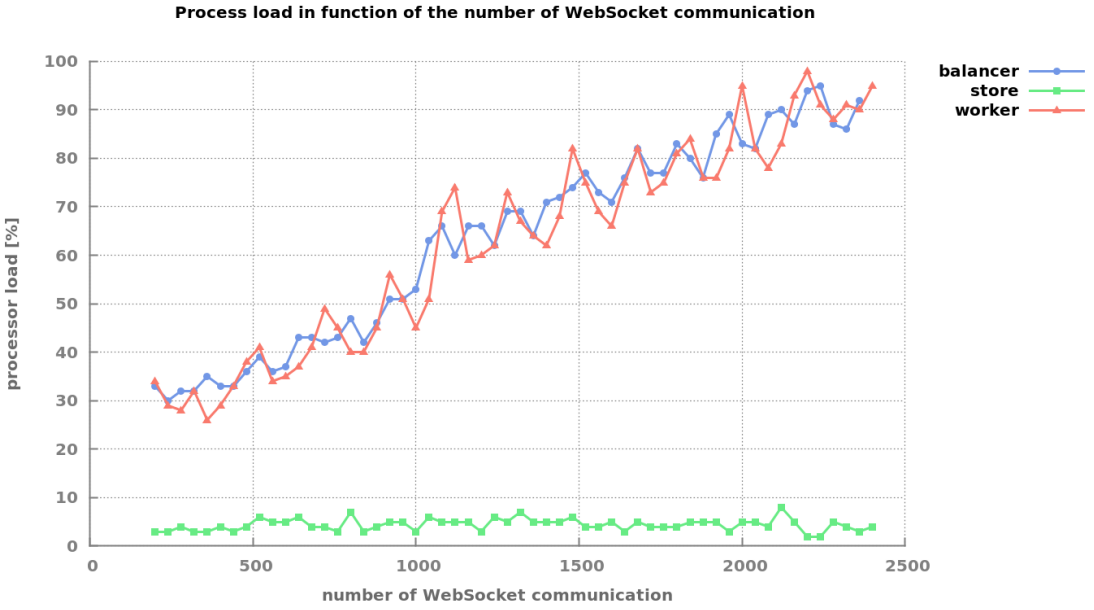

FIGURE 4.7: Server with 3 cores

figure 4.7 clearly shows the worker and loadbalancer cores are almost used to their full extent. In order to handle more communication more cores should be added.

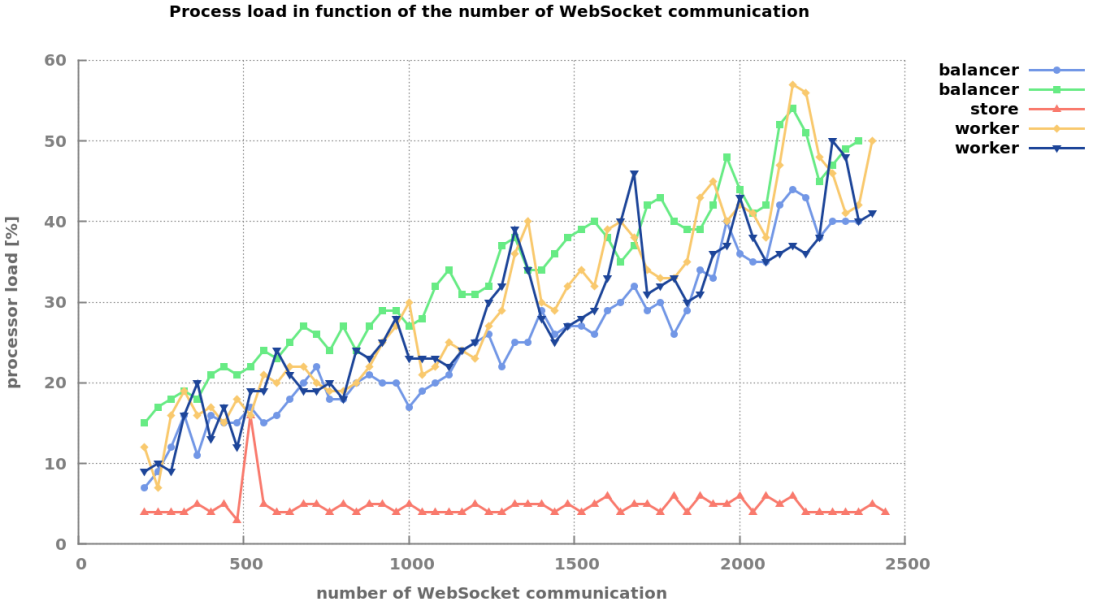

FIGURE 4.8: server with 5 cores

In this experiment two more cores have been thrown to work. Load balancers and workers nicely balance the work between themselves and the maximum load drops to 50%.

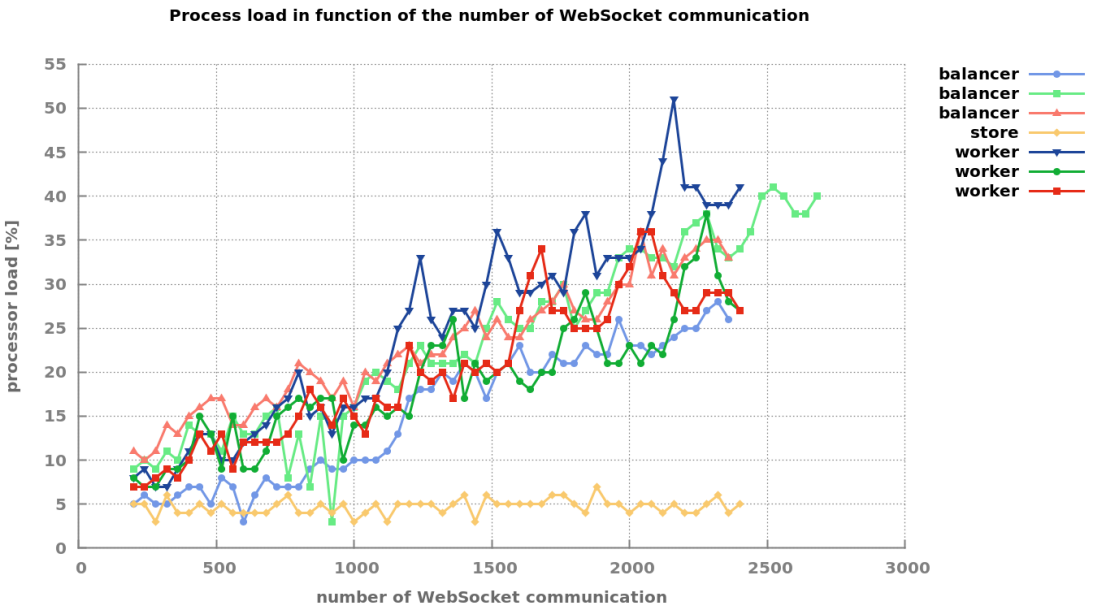

FIGURE 4.9: server with 7 cores

this last test is less conclusive. With a total of 3 cores for load balancers and three for workers the processors load varies between 30% and 50% depending on the task.

Adding too many cores is a waste of ressources, this stresses the importance of finding a load balancer/worker/store ratio rule.

B] section most influence frequence pings / size message / number communications

c] General rule

load balancer / store / worker

Appendix A

SocketCluster

A.1 Simple ping-pong exchange

Client 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: Pings from client

To best simulate clients interaction with a websocket server, new sockets are created at random intervals `intv = Math.round(Math.random()*5000)`.

Server code

The server listens for pings event and answers back with pongs event. In this case the pong event is an integer counting the number of pings this particular worker had during the whole experiment.

```
// Handles incoming WebSocket connections and listens for events
wsServer.on("connection", function (socket) {

  // The server listens to the ping event from the clients
  socket.on("ping", function (N) {

    // Sends simple pong event
    socket.emit('pong', ++count);
  });
});
```

FIGURE A.2: Server answering with pongs

A.2 File transfer

Client code

In this example, the goal is to exchange a file using the WebSocket protocol. For this purpose, the node.js `delivery` library is used.

New clients are created on the same model as the previous exmample. Each new client is stored in the `clients` array. Each clients are also perdiodically sending pings. The only add on is the `map` function to enable the each socket to retrieve the document sent by the server.

```
var dl  = require('delivery');

clients.map(function(client){

  // RECEPTION OF FILES
  var delivery = dl.listen(socket);
  delivery.on('receive.success',function(file){
    fs.writeFile(file.name, file.buffer, function(err){
      if(err){
        console.log('File could not be saved: ' + err);
      }else{
        console.log('File ' + file.name + " saved");
      };
    });
  });
});
```

FIGURE A.3: Clients receptionning files

Server code

The server listens for pings. And sends back a file, `foo.txt` in this example.

```
var dl  = require('delivery');

// Handles incoming WebSocket connections and listens for events
wsServer.on("connection", function (socket) {

  // The server listens to the ping event from the clients
  socket.on("ping", function (N) {
    var delivery = dl.listen(socket);
    delivery.connect();
    delivery.on('delivery.connect',function(delivery){

      delivery.send({
        name: 'foo.txt',
        path : './foo.txt'
      });

      delivery.on('send.success',function(file){
        console.log('File successfully sent to client!');
      });
    });
  });
});
```

FIGURE A.4: Server sending files

Appendix B

Engine.io

This appendix gives the code used to create a simple engine.io server and client. Comparaison with SocketCluster code in appendix A shows the difference between both implementation is small.

In fairness, SocketCluster API is very close to engine.io.

Client code

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

if (cluster.isMaster) {
        for (var i = 0; i < numProcs; i++) {
                var worker = cluster.fork();
        }
} else {
        var connectSocket = function () {
    var destination = 'http://' + hostname + ':8080';
    var socket = require('socket.io-client')(destination);
    var intv = Math.round(Math.random() * 5000);

    // SENDS PINGS
    setInterval(function () {
      socket.emit('ping', {param: 'pong'});
    }, intv);
    setTimeout(connectSocket, 1000/numberClientsEachSecond);
  };
  connectSocket();
}
```

FIGURE B.1: Pings from client

Server code

```
var app = require('http').createServer(handler)
var io = require('socket.io')(app);
var count =0;

app.listen(8080);

io.on('connection', function (socket) {
  socket.on('ping', function(N){
    socket.emit('pong', ++count);
  })
});
```

FIGURE B.2: Server answering with pongs

Appendix C

Real time throughout check

By inserting the following script in `index.html` the browser will display in realtime the number of pings received by a WebSocket server.

```
<script type="text/javascript">
  var options = {
      protocol: 'http',
      hostname: "localhost",
      port: "8080",
      autoReconnect: true
  };
  var socket = socketCluster.connect(options);
  socket.on('connect', function () {
    console.log('CONNECTED');
  });
  socket.on('pong', function(data){
      console.log(data);
    var curHTML = document.body.innerHTML;
    curHTML += 'Number of pings: ' + data + '<br />';
    document.body.innerHTML = curHTML;
  });
  socket.emit('ping');
</script>
```

FIGURE C.1: Modification to `index.html`

All it does is emitting a ping, then listening to the pong event and displaying it directly in the html page. The pong payload as can be seen in A.2 is `count`, an integer increamented each new ping.

Bibliography

[1] J. Mogul R. Fielding and J. Gettys. Hypertext transfer protocol – http/1.1. *Request for Comments 2616*, 1.4 Overall operation, September .

[2] Eliot Estep. Mobile html5: Efficiency and performance of websockets and server-sent events. *Master thesis*, 3.3 Web techniques, June .

[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 .

[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 .

[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 .

[11] Mikal Bourges-Svenier. Graphics programming on the web, webcl course note. *Special Interest Group on GRAPHics and Interactive Technique conference*, October .

[12] Tasneem Brutch Won Jeon and Simon Gibbs. Webcl for hardware-accelerated web applications. *Tizen developer conference*, May .

[13] A. Barak and A. Shiloh. The virtualcl (vcl) cluster platform. *Mosix white paper*.

[14] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, pages 483–485.

[15] Nathan T Hayes. High performance parallel computing in the motion picture industry. *Sunfish white paper*, February .

[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 .