Report

Large Scale Web Chat Application

Semester Project

Author: Tugdual de Kerviler
Supervisor: Nicolas Bonvin
Professor: Karl Aberer

June 11, 2010
## Contents

1 Push Technologies for Web applications 3
   1.1 Concepts ................................................. 3
      1.1.1 Comet .............................................. 3
         1.1.1.a Simple polling .................................. 4
         1.1.1.b Long polling .................................... 5
      1.1.2 Comet without AJAX ................................. 6
      1.1.3 HTTP Streaming ...................................... 6
      1.1.4 Comet with HTTP Streaming .......................... 7
      1.1.5 HTML5 WebSocket .................................. 7
      1.1.6 Applets .............................................. 8
   1.2 Technologies ........................................... 8
      1.2.1 Push Engines ....................................... 9
      1.2.2 Server ............................................. 10
      1.2.3 WebSocket ......................................... 10

2 Architecture .............................................. 13
   2.1 Node.js ................................................ 13
   2.2 MongoDB .............................................. 18
   2.3 Architecture scheme .................................. 19
   2.4 Architecture scalability ............................... 20
   2.5 Usercase ............................................... 20

3 The chat application .................................... 21
   3.1 Getting started with Node.js ........................... 21
   3.2 Coding ................................................ 21
      3.2.1 Javascript ......................................... 21
      3.2.2 Git ................................................ 22
   3.3 The application ....................................... 23

4 Application benchmark .................................. 24
   4.1 Benchmarking tool ..................................... 24
   4.2 Results ............................................... 24
   4.3 Links .................................................. 28
Introduction

This report presents an overview of my semester project which happened during the spring semester in the Distributed Information Systems Laboratory at EPFL. The project was consisting in a research in push technologies and in a practical application with the implementation of a chat.

1 Push Technologies for Web applications

1.1 Concepts

Web applications works onto the HTTP which is client-server, this means the client initiate the connection, the server only responds to its request. With the evolution of the web, people have wanted applications to be more interactive and to be able to give live information to their users. We are going to look at the different mechanisms to achieve this technique which is called server push.

1.1.1 Comet

Comet is a term invented by Alex Russell of the Dojo project who was actually joking about AJAX cleanser. The technology is known by several other names like Ajax Push or Reverse Ajax but only refers to the possibility of a server to "initiate" the communication in a web application using AJAX. Thus, in this part we will only consider push using AJAX.
1.1.1.a Simple polling

It can be thought as automating the refresh of a page (or a part of it). To achieve it, a client can simply do periodical AJAX requests to the server. The server always responds to it immediately and when it has nothing new, it simply sends an empty message. In this implementation, it is important to choose well the delay between two requests. If the delay is too short, messages will saturate server’s connection, if too long, the interest for the technology is decreased.

This solution has many advantages:

**Compatibility**: with every browser which supports JavaScript and implement XMLHttpRequest (so most of them).

**Easy to use**:

- **On server-side**: a classic HTTP server is enough.
- **On client-side**: the Javascript code is not very complicated.

- But has also many disadvantages which are waste in:

**Bandwidth resource**: the numerous empty messages greatly decrease the throughput.

**Processing resource**: for the huge amount of requests which have to be processed in a short time.
1.1.1.b Long polling

An evolution of simple polling can be thought by remembering the beginning of the Internet, when pages were loading slowly. Thus, we can build a special server which does not respond until it has the information. From the point of view of the client, it seems that the server is lagging but in fact this is the key idea of Comet and it works pretty well!

+ Advantages:

  **Compatibility**: as well as for simple polling, the client just need a browser which supports AJAX.

  **Scalability**: bandwidth resource is not wasted anymore.

- Disadvantages:

  **Harder to implement**:

  **On the server**: we need software modifications. A very problematic one is certainly the “C10K problem" which is a due to the OS constraint and software limitations. When a server which implements long polling receives more than ten thousands requests, if it uses default blocking IOs, it will have to keep in memory ten thousands threads. This consumes lot of resources in RAM memory can over ow. Thus classical servers are now implementing powerful tools such as Asynchronous Request Processing (ARP) for Java Servers which permits non-blocking IO.
On the client: browsers usually implement a timeout for a request, so the client program needs to handle this. A simple way is to forget the query after a certain amount of time (less than the timeout) which means that even if the server responds after that time, the response is not processed.

1.1.2 Comet without AJAX

There are other ways to achieve push without using AJAX. One of them is called HTTP streaming.

1.1.3 HTTP Streaming

HTTP Streaming uses the idea of persistent connection introduced by HTTP. The mechanism which achieves it is called chunked transfer encoding. It allows HTTP messages to be split into several parts in both directions. The classical way to use it is to insert an invisible HTML IFrame in the page which content is sent in several pieces thanks to this mechanism. This way the page is loaded first time and then updates are done reading the IFrame which continuously comes in. Pieces sent must be JavaScript code included into the HTML script element.
+ Advantages:

**Throughput** is excellent as no message is useless.

- Disadvantages:

**Firewalls an Proxies**: often blocks streaming.

**Unreliable**: if a user does hit the stop button, all of the streaming content will be killed - unbeknownst to the user.

**Timeout limit**: which is imposed by the browser (5 minutes by default on firefox). After this limit, we need the user to refresh the page manually.

**HTML dependent**: the method is not really clean as it relies on the IFrame HTML element. We can, for instance, think of a browser extension which blocks IFrames (like AdBlocks).

**Structureless**: by opposition to AJAX which uses clean XML messages, here the server has to generate Javascript code to modify the page and include it into HTML script elements. Moreover, we have an IFrame which content grows indefinitely.

### 1.1.4 Comet with HTTP Streaming

Mixing the two technologies is very beneficial. It increase Comet’s throughput and many of the disadvantages of using HTTP streaming alone no longer exist! In this implementation, we do not use any IFrame. It works the same way as long polling, however, the server keep the connection alive after it respond. When the browser timeout, AJAX client re-establish the connection.

+ Advantages: Same as for long polling. Throughput is better

- Disadvantages: Same as for long polling. This technique is complicated to implement so the best way is to use a Push Framework. Firewalls and proxies issues.

### 1.1.5 HTML5 WebSocket

The WebSocket API is an interface defined by HTML5. This means that browsers which want to fully support HTML5 will need to implement it. It permits HTML5 webpages to open a socket connection with the server and thus allows everything which is possible with a classical TCP/IP connection. This method is by far the better as it has been invented for this purpose. However, old browsers do not support HTML5 and this technology is still experimental.
1.1.6 Applets

We have Java applet, Flash and other browser plugins. We will not be talking about them as their technologies requires additional components to the web browser and are often proprietary. Just to say that they have been used for bidirectional communication on the web for a long time (before AJAX became popular) and are still used for many applications. But when it comes to change the content of a page, these technologies are not really appropriate.

Sources:

- Comet and Reverse Ajax: The Next-Generation Ajax 2.0 By Dave Crane and Phil McCarthy
  - [http://weblogs.java.net/blog/jfarcand/archive/2007/05/new Adventures.html](http://weblogs.java.net/blog/jfarcand/archive/2007/05/new_adventures.html)
  - [http://ajaxpatterns.org/HTTP_Streaming](http://ajaxpatterns.org/HTTP_Streaming)

1.2 Technologies
### 1.2.1 Push Engines

<table>
<thead>
<tr>
<th>Description</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orbited It enables TCP/IP-based protocol to run over HTTP. The JavaScript client is mainly shaped to parse TCP/IP packets.</td>
<td>MIT</td>
</tr>
<tr>
<td>Ape-project Full solution to build a Push application. The server is written in pure C. The client framework is compatible with famous AJAX libraries such as jQuery.</td>
<td>GPL</td>
</tr>
<tr>
<td>Cometd This project of the Dojo Foundation proposes implementations of the bayeux protocol in Java, Python and Perl. Bayeux protocol has been invented to simplify Push applications development. It especially solves common message distribution and routing problems.</td>
<td>Apache+EPL</td>
</tr>
<tr>
<td>Meteor This Push Engines provides a webserver written in Perl and a JS class for the client.</td>
<td>GPL</td>
</tr>
<tr>
<td>LightStream This solution is known to be very impressive and is not free.</td>
<td>-</td>
</tr>
<tr>
<td>Grizzly This project offers multiple frameworks for Push on Glassfish and Jetty web servers. It uses the &quot;new and non-blocking IO&quot; API (NIO) to implement ARP for high scalability on Java servers.</td>
<td>CDDL+GPL</td>
</tr>
<tr>
<td>Atmosphere It is an evolution of grizzly which is compatible with every java-based webserver. It supports Servlet 3.0 (for ARP) as well as Cometd.</td>
<td>CDDL+GPL</td>
</tr>
<tr>
<td>ICEfaces This Java solution provides a very complete AJAX application framework. It proposes a huge amount of UI components and libraries to build a professional web application very easily. Lots of them are not free!</td>
<td>MPL</td>
</tr>
<tr>
<td>ZK This is a rich framework designed to build professional mobile applications.</td>
<td>GPL/Comm.</td>
</tr>
<tr>
<td>Lift This web framework is written in Scala. Thus, compatible with Java libraries. It claims to provide the same developer productivity as Rails!</td>
<td>Apache</td>
</tr>
</tbody>
</table>
### 1.2.2 Server

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Apache</strong></td>
<td>This famous HTTP server works with modules which allows many languages.</td>
<td>Apache</td>
</tr>
<tr>
<td><strong>Glassfish</strong></td>
<td>This open source Java application server is a complete implementation of Java EE developed by Sun. It has developed the powerful Grizzly project.</td>
<td>CDDL + GPL</td>
</tr>
<tr>
<td><strong>Jetty</strong></td>
<td>This is a flexible Java webserver which implements ARP. It is recommended for high scalability. For instance, Google AppEngine uses it for its Java service.</td>
<td>Apache</td>
</tr>
<tr>
<td><strong>Tomcat 6</strong></td>
<td>This well-known server supports Apache Portable Runtime (APR) and NIO APIs. It now provides classes to build Comet applications.</td>
<td>Apache</td>
</tr>
<tr>
<td><strong>nginx</strong></td>
<td>This is an asynchronous webserver. Like Apache, it works with modules. It is very used for Ruby on Rails websites.</td>
<td>BSD like</td>
</tr>
<tr>
<td><strong>Tornado</strong></td>
<td>Tornado is a Python webserver with non-blocking IOs.</td>
<td>Apache</td>
</tr>
</tbody>
</table>

### 1.2.3 WebSocket

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kaazing Open Gateway</strong></td>
<td>This is the open source version of the famous WebSocket server which is probably the most complete one today.</td>
<td>CPAL</td>
</tr>
<tr>
<td><strong>node.js</strong></td>
<td>node.js is brings asynchronous IOs into Javascript. Multiple module have been developed to handle websocket</td>
<td>MIT+GPL</td>
</tr>
<tr>
<td><strong>jWebSocket</strong></td>
<td>This project provides a light stand-alone Java WebSocket Server and a JavaScript WebSocket client.</td>
<td>GPL</td>
</tr>
<tr>
<td><strong>WebSocket in Tornado</strong></td>
<td>A Tornado module for WebSocket has been developed.</td>
<td>Apache</td>
</tr>
</tbody>
</table>
Now let’s see interesting implementations for each technology:

**Simple polling** can be implemented on any server without any Push framework. An Apache HTTP server (+FastCGI) with any language program is a good solution.

**Long polling** is more complicated as it needs to handle a lot of users at the same time. A Push engine associated with a webserver is a good solution to develop a simple application.

**Orbited and Ape-project** are very flexible and easy solutions. Associated with a classic Apache HTTP server for serving webpages, we should obtain good performances.

**Lift** is very interesting if we want to program fast web applications in Scala. It can be used with any Java-based webserver such as Tomcat.

**Cometd** has the advantage to implement Bayeux protocol. This solution should really be considered for very large applications. The best way to use it is with the Jetty webserver.

**Tomcat + Atmosphere** is certainly very good for a solid Push application. It allows Java-based applications maintainability, Tomcat’s automatability and Atmosphere’s flexibility.

**Icefaces** provides an asynchronous server implementation but can also work with Jetty or GlassFish. It can be envisaged for applications providing professional services where it really seems to be a turnkey solution.

**Juggernaut** is a push engine for Ruby on Rails.

When looking for performances we might want to look at event-based webservers:

**Nginx** is the most famous (and probably the fastest) event-based webserver.

**lighttpd** (+FastCGI) is another great webserver. It is often used with the Python language.
node.js is a new Javascript server-side framework which is currently in active development and for which many people see a great potential. It provides asynchronous IOs in Javascript and run with Google V8 engine.
2 Architecture

After this research, Nicolas convinced me to try node.js for this project. It is really interesting to work on new technologies and I thought I will be able to learn a lot about webservers, scalable architectures and NoSQL databases. Moreover I was not really good in Javascript so it was an opportunity for me to learn this language which is becoming more and more popular.

2.1 Node.js

The aim of the project is to build a large application chat using brand-new technologies. Node.js is one of them. It has a great potential and people are going to hear a lot about it in the next few years.

Node is a recent project written in C and JavaScript and using Google V8 Engine which allows developers to write event-based server-side applications in Javascript.

But first, let's see why this solution is better than using a classical thread-based webserver like Apache.

In a webserver using threads, when the client requests a page, the server creates a new thread to process it separately. This way it will be able to listen for new requests at the same time. So the architecture is one thread per client. A thread runs until the page is generated. Often, the page is loaded using a request to the database which does not respond instantaneously. During this period of time, new users can request a page and thus new threads will be created and db queries will enqueue. Here, we can suppose the time period is short, less than 1 second. So the number of possible users at the same time is limited and it works pretty well.

But what happens if we want to build a Comet application? i.e. an application which keeps the connection open between the server and the client for a long time. Here, the resource each thread is waiting for is not some hardware fast stuff anymore, like reading on the hard drive or requesting a local database server. It is an operation which can be very long, like a user response in a chat application, the evolution of stock prices in a stock market application, a request on a server working with tapes...

If we want 100,000 users at the same time on the application, we will need to have at least 100,000 threads in our system and we will quickly come to...
system limitations (both software and hardware). This is not the right way for building Comet applications today.

We want linearly scalable architectures. We want the code to be simple and effective. From this needs, many servers have emerged. They replace threads model by events’. In this model, we have only one thread for the main server program and maybe one thread for each ressource. When a ressource is available, the thread attached to it sends an event to the main program which then handles it. As the number of thread is minimal, such a server requires a lot less memory. Moreover, processing a smaller amount of data often involves a gain in time processing. This model is thus a lot more scalable. The most famous webserver using it is certainly nginx. A benchmark is often presented to show its performances against Apache:

Request per seconds comparison
With the emergence of dynamic websites and AJAX technology, JavaScript has become the language that cannot be ignored on the web. Google has seen it and has developed V8, a Javascript engine with enhanced performances. The JavaScript language has been created to handle events easily so it is in fact particularly well adapted on the server-side. Javascript has 2 very interesting functionalities: Anonymous functions and Closure. Those really allows developers to keep the code simple when writing asynchronous functions:

**Anonymous function:**
In Javascript functions can be declared as variables:

```javascript
// declaring an anonymous function
var square = function(x) { return x*x ; }
```

This is very usefull when writing asynchronous code as it permits to declare a function which accepts another function as a parameter. This function will be able to call back this parameter function when it has finished its job. So for example:

```javascript
fs.readFile("myfile.txt", function(err, data) {
   if(err) throw err;
   // print content
   sys.puts(data);
});
```

while readFile could look like:
// file fs.js
exports.readFile = function(filePath, callback) {
    if (fork() === 0) { // create new process
        try {
            f = open(filePath);
            buff = "";
            while ((byteRead = f.read(1000, buff)) !== 0) {
                callback(null, buff);
            }
        }
    catch (err) {
            callback(err, null);
        }
    }
}

This is very interesting for a server as we often need to access the database for instance when generating a page. The database driver query function should accept as a second argument a function which is called when the result of the query has been received.

An abstraction of this functionality is the Event model: When working with events one can create an Object which is able to notify you by an Event when something new happened.
In fact, this is already used in every browser. Every HTML page has a document object which is attached to it and which can be used by the page itself to define functions which must be called when an event is triggered.
For example:

document.load = function(event) {
    // the page is loaded let’s execute some scripts
}

The browser has probably a few threads which are watching for user interactions. Each time the user moves his mouse the browser is notified by the thread and calls the function to the onmousemove variable. The event parameter will contain any information related to the event. In our case, the position of the mouse.
This model provides a clear and simple interface for developpers which are using an API. And it can be very usefull when writing a large application.
Closure:

Closure is a functionality that allows a variable to be spread when a new context is created. It is related to the scope of its variables. Every function creates a new context when it is defined. The free variables of this context are the the parameters of the function as well as the variables created on by any parent context. For example:

```javascript
var sys = require("sys");

// context 1
x = 0;
function a() {
    // context 2 defined after x = 0;
    // x == 0
    y = x+1; // y == 1
    var b = function(y) {
        // context 3 defined when a is called
        // x == 1
        // y == this.arguments[0] == x == 1 // first argument
        return y+x;
    };
    return b(x);
}

x = 1;
sys.puts(a()); // prints 2
```

Let’s see how it is useful on real cases. First it decreases a lot the number of parameters a function should have and simplifies the code. Let’s get back to our readfile function: We want to define what should be done when the file has been read. This function has been written by someone else in a seperated library so there is no question to modify it. Then how could I use information I have about previous execution of the program. For instance, without closures, this would not be possible:

```javascript
function parse(str) {
    // function which parses a string
    return str.split("","");
}
```
function readAndParse(file, callback) {
  fs.readFile(file, function(err, data) {
    if(err)
      callback(err, null);
    else
      // return parsed content
      callback(null, parse(data));
  });
}

readAndParse("myfile.txt", function(err, results) {
  if(err) throw err;
  else
    // process results
});

So it’s really a useful functionality which is used all the time when writing a node application. Closures are also used to implement continuations which briefly permits to generate functions.

2.2 MongoDB

MongoDB is a one of these new NoSQL database everyone is hearing of and would want to try. It is not a relational SQL database but a Document Oriented one. Documents are saved using the BSON format (Binary JSON) and are retrieved in JSON using hash references. MongoDB has very cool features for querying which are in fact Javascript objects. So this is really suited for node.js. But more important, MongoDB is fast and highly scalable. This means we can easily make a cluster which every servers can query. Using such a database greatly simplify the conception of a scalable architecture. It comes with a very interesting distributed file system called GridFS which can be used to store files.
2.3 Architecture scheme

In a first time, we are thinking of using only node. Even if it is a very young project, it is very active and has already many interesting modules which should be enough to build our entire application.

Full node.js architecture
2.4 Architecture scalability

MongoDB has a scalable architecture: it can replicate on multiple servers to form a database server cluster. Multiplying the number of machines in the cluster by 2 and the cluster should be able to support the double amount of requests. Our application is linearly scalable too! It means if we need to support the double amount of users we simply double the number of machines. The DNS server should have a load balancing solution to redirect users to one of the node servers. A room exists on only one server at a time. If a user requests a room which is on another server, the current server will ask the database which server is hosting the room and redirect the user to it. So once the user are connected to a room, there is no interactions with other node servers. It behave like if there was one server handling a smaller amount of users.

2.5 Use case

2) The DNS server returns default server’s IP (here Server 2).
3) Client A connects to Server 2 and request a new room.
4) Server 2 redirects Client A to Server 1 which will host the room (redirected to http://www1.chat.com/idroom).
5) Server 2 add a new room entry in the database and send back the chat application.
6) Client A initialize a bidirectional communication with Server 1. It is connected to the chat!
7) Client A gives Client B the url to join the chat room.
8) Client B connects directly to Server 1 thanks to the www1 prefix.
9) Server 1 add a new user for the room and sends the application.
10) Client B initialize a bidirectional communication with Server 1. It is connected to the chat!
11) Client A and B receive each others messages which are saved in the database for a few hours.
3 The chat application

3.1 Getting started with Node.js

Node.js is developed by Ryan Dahl but has also several contributors. It is still experimental but already shows its great performances. A lot of node projects are on github as well as node itself. The wiki page lists many useful modules.

3.2 Coding

3.2.1 Javascript

I really have learnt a lot about Javascript during this project. It is in fact quite a cool language. It is really simple to use callbacks for asynchronous functions as soon as we have understood the basics. It is very different from usual programming. At the beginning I tried to apply my knowledge of OOP and tried to avoid using prototype so I used a Class module to see everything as an object. However I had many difficulties in the conception of the program and while I was chatting with node users talking they confirmed that this is not always a good solution and that we should not always use objects. When working asynchronously it can become very tricky to mix callbacks with objects. I also learnt how to use git as I have hosted the project on github too.

Node.js tries to implement the CommonJS interface. CommonJS defines many conventions programmer should agree on when writing Javascript libraries. There are two conventions any programmer which starts with node should be aware of:

- The callbacks’ arguments format:
The first argument of the callback should be an Error object (null if no error):

```javascript
callback = function(err, arg1, arg2, arg3)
```

- The module format:
To access to a variable from another module one must call require(module_path).
In the required module every public elements must be added to the special variable exports:

```
// in math.js
exports.square = function(x) { return x*x; }  
```
var square = require("./math").square;
sys.puts(square(3));

3.2.2 Git

I never used a software repository before and I thought I had to start with git as a lot of node projects are on github.com. It is full of features and it is really not easy for a beginner to get started. Hopefully, we have little tutorial when creating a repository on github which was very helpful. After creating your repository, you need to add your public key on your github account to be able to use it. You first need to set the globals:

```bash
git config --global user.name <your_github_username>
git config --global user.email <your_sshkey_email>
```

After this, you can create your working folder and init git:

```bash
mkdir express-chat
cd express-chat
git init
```

Then create some files and add them to the repository:

```bash
git add .
git commit -m "What has been modified"
git push origin master
```

These last three operation are to be repeated each time you want to update the repo.

When working with several computers, you may want to update the code you pushed from another computer. For this just type:

```bash
git pull
```

After cloning a repository, you may want to come back to an earlier tag version:

```bash
git checkout <tag_name>
```

Submodules:

- adding a submodule:

```bash
git submodule add git://github.com/<creator_name>/<project_name> <submodule_folder_path>
```

When adding a submodule, it is synchronized to the last commit by default, so even if the project is updated it will clone the right commit. When cloning a project which has submodule, you have to init the submodules:
git submodule update --init

- Removing a submodule is a bit more complicated:
  remove manually the corresponding submodule files in .gitmodules and .git/-
  config files, then
  git rm --cached <submodule_folder_path>
There is no simple way I know to update a submodule, so each time I want
  to update, I have to remove it and add it again.

3.3 The application

The application is a simple chat. The main program is contained in two files
(app.js and room.js). app.js defines the instructions to be executed for each
type of request. It create room objects which definition is in the room.js
file. Each rooms use three EventedBuffer objects to handle messages, users
and files. EventedBuffer is in fact a buffer for which we have associated
four events: added, removed, changed and all. Each corresponding event is
triggered when an element is added, removed or changed in the buffer. For
this, it defines a addListener(event, callback) method which save the callback
and calls it when the event happens. This 3 files are almost enough for having
the chat working. However we want it to run on multiple server. This is
why we have choosen to use a database. To connect to mongoDB we must
use a driver. These are in active development and evolves with mongoDB
itself. The most advanced one is probably the node-mongodb-native one
which is entirely written in javascript. But functions it provides are in fact
pretty basic so I had to write wrappers to easily access my objects in the
database. This is the purpose of all the files in the mongo folder. Today,
the mongoose project allows the creation of simple wrappers cool syntaxes.
However it was not developped when I started my chat. The native driver
offers bad performances when using GridFS. After many tries at increasing
the performances I chose not to use it for files and simply called the binary
command-line driver shipped with mongo.
4 Application benchmark

4.1 Benchmarking tool

Now that the chat is done we want to test the scalability of our architecture. What is interesting in our application is to see the performances for long polling requests. So, we want to measure the delay a message takes when sent by some user and received by every other on the chat room. For this I needed a framework (or a rich tool) in order to mark every messages and then deduce the time. I came across multimechanize which is a load testing framework working with the mechanize library on Python. Mechanize has an interesting Browser object which permits to navigate on a website like a real browser (except it does not execute Javascript). It supports cookies which is useful to distinguish users without modifying the whole application and redirecting which is useful when testing with multiple servers. However, I found the multimechanize not performant both in usability and in fastness. So I had to simplify it for my purpose. For instance, I removed the eval() in each thread and added a lock feature. I tried to avoid using the Browser functionality when possible to gain performance so I allowed a room to be created from a simple request. During the tests, Browser is only used to send messages from the chat page. This way every user is identified from his session id.

4.2 Results

I have been trying to bench the average time between the sending and the receiving of a message. For this, each time we create a room, we add 5 users in it. Four are senders and one is a listener. Tests were not very good at first sight. Two servers showed about the same response time as one. Sometimes results were even worse:
Average response time between the send of a message and its reception

But when looking at the throughput, we can clearly see that it is better with two servers:

Throughput for one client requesting one server
Throughput for one client requesting 2 servers

How is it possible?
In fact results do not show requests that are not finished yet. Thus, requests which have not been taken into consideration by the server simply does not appear in the results. I did not have time to rewrite the stress-test to make those results appear, but the facts are here:
One server has responded to 174358 requests while for the same load 2 servers has responded to 320620 requests. As an HTTP connection is reliable, it means requests which have not been served yet are from rooms that have not been created. Thus, we can simplify our graph by setting the maximum number of room to the corresponding:

\[ \sum_{i=1}^{n} 5n = 320620 \Rightarrow n = 357 \]

and scale down the results for one server to:

\[ \sum_{i=1}^{n} 5n = 174358 \Rightarrow n = 263 \]

And we obtain the attending graph:
Real average response time between the expedition of a message and its reception

Real response time at 90% between the expedition of a message and its reception
Here are the raw data for one (of the four) client requesting one server:
For all requests (loading of the pages and exchange of messages):

For messages only:

4.3 Links
My repository: [http://github.com/Tug/express-chat](http://github.com/Tug/express-chat)
Node API: [http://nodejs.org/api.html](http://nodejs.org/api.html)
Node google group: [http://groups.google.com/group/nodejs](http://groups.google.com/group/nodejs)