UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Computer Science

Mihkel Jõhvik

Push-based versus pull-based data transfer in AJAX applications.

Bachelor's thesis (6EAP)

Supervisor: Peep Küngas, PhD

Author: ..... „...“ June 2011

Supervisor: ..... „...“ June 2011

Approved for defense:

Professor: ..... „...“ June 2011

Contents

Abstract

The Internet contains an ever increasing number of interactive applications, powered by technologies like AJAX and COMET. The latter has multiple ways of achieving the desired functionality, namely long-polling and streaming. While there have been many comparisons between AJAX and the traditional model of the web and even AJAX and long-polling, little research has been done comparing streaming to long-polling. This thesis aims to compare the different technologies in a framework created by Ali Mesbah et al., who compared polling and long-polling solutions in their 2008 article „Performance Testing of Data Delivery Techniques for AJAX Applications [2]“. A new streaming test was created and compared with the other tests included in the framework.

The results showed that streaming used very little network resources, using only one packet per message after the initial connections were made. Latency was also not an issue as messages were received by the client in 1305ms on average. However CPU usage and data coherence were not as good. Streaming dropped 1-2 messages where long-polling dropped none and the CPU usage was high at unpredictable times.

These results, along with other shortcomings of the streaming method, led the author to determine that streaming is not a viable alternative to long-polling or polling at this time.

1. **Introduction**

There has been a notable push towards more interactive web applications in recent years. Many web services, such as Google's Gmail, now offer the same functionality that desktop programs do, the only restriction being Internet access. One of the things that have made this possible is AJAX, a term used to describe a web programming technique where small amounts of data are traded between the server and client asynchronously. Usually, a small JavaScript script is used to connect to the server using an object called the XMLHttpRequest[1] (also called XHR, in this thesis these terms will be used interchangeably). This object has the necessary methods and parameters to connect to the server and hold the server response, all without reloading the entire page the script was called from. The server response can then be manipulated or inserted into the DOM using traditional JavaScript methods [6][16]. This has greatly increased the responsiveness of web pages and allowed programmers to take on a whole new array of tasks that were previously inconceivable.

However as the web grows, so evolve the demands it makes of its applications. *Latency*, *data coherence* and *data redundancy* are important factors in how the user perceives web applications and in some occasions, data coherence is extremely important. Latency is the time it takes for an event that occurred on one side of the connection to register on the other side [2]. From a user's stand point it usually means notification of the registered event. For example in a chat application, when a user hits „send", their entered message will show up in the chat window.

Data coherence means that the client- and server-side data share the same state. When a change occurs on one, the other is immediately updated [2].

Data redundancy will occur when the same data is transferred to the client more than once, when there have been no errors on the receiving end. This means that the client already has the necessary data, but the same data is transmitted again, causing unnecessary network traffic [2].

Some examples of applications that might need such data coherence and low latency:

- Stock tickers – tickers which show the value of stocks in real time.
- Auction sites – websites where multiple people can bet on various items. To see if anyone else has bid on the same object, the user does not have to refresh the web page.
- Sports coverage sites – websites which cover sports events. To see various updates (goals, standings), the user does not have to refresh the web page.
- Banking sites – the websites of banks which allow the user to transfer and receive funds. To see if their transactions have gone through, the user doesn't have to refresh the web page.

[1] http://www.w3.org/TR/2010/CR-XMLHttpRequest-20100803/

- Communication applications – instant messengers like Facebook Chat[2] or Google Wave[3], which allow users to communicate with each other in real time.
- Auto-complete applications – applications which suggest auto-complete options for users filling out forms. An example can be found in the Google[4] search engine's auto-complete function.
- Applications that manipulate the data before showing it to the user. For example, a diagram application that updates its contents regularly in a live environment.

If the user does not have the most up-to-date data then errors can occur. The latency in receiving data could also cost a company running such a program some of its revenue as users look for latency-free alternatives. An example of an error is if someone wishing to bid on an item that is being sold on an auction website doesn't see that someone has already bought the item. This might lead to erroneously crediting the user's account, marking the item as sold to the user when in fact it hasn't been and so forth. While safe programming practices and proper system architecture should not allow for these types of mistakes, there is a chance that something does go wrong and that may cost the web application a user, if not more, and the company running that application its reputation.

To handle these types of problems, different solutions have been devised. The first one, polling, uses AJAX and the ability of JavaScript to set a time delay for executing a command to frequently ask the server if new information has been made available [1]. Long-polling is an evolved variant of polling where the connection is not closed by the server when no new info is present, but instead when a timeout occurs or when new data becomes available. A new request is initiated right after the first one is received [1]. The third, called streaming, uses an indefinitely held HTTP connection to allow pushing new information by the server to the client when it becomes available [1]. The last two methods are also known under the umbrella term COMET which was coined by Alex Russell and is used to refer to any method where the server is pushing out data when it becomes available instead of when a client asks for it [15]. A more detailed explanation of these methods is located in the background section of this thesis.

The goal of this thesis is to ascertain whether there is a commercially viable solution to streaming and to compare streaming to long-polling and polling solutions created by Engin Bozdag, Ali Mesbah and Arievan Deursen in their article „Performance Testing of Data Delivery Techniques for AJAX Applications" [2]. A plug-in to ease COMET development for the JavaScript library jQuery[5] will be created. This thesis aims to answer the question: which solution is better to use in an application that is open to the public?

[2] http://www.facebook.com/sitetour/chat.php
[3] https://wave.google.com/
[4] http://www.google.com/
[5] http://jquery.com/

2 Background

2.1 Polling

The ability of AJAX to load only a part of a web page without refreshing the entire website can be used to create dynamic web applications that more and more resemble desktop applications. However, AJAX still follows the current HTTP request-reply architecture, which expects a reply or an error notification for each request [14]. This does not include long-lived connections, so when the application requires serving the user with constantly changing data, an AJAX script has no way of knowing whether the data on the server has changed or not. It must then keep constantly querying the server whether new data is available or not [13]. This is called polling, short polling or pull [9]. The client sends a request; the server processes it and then sends a response back. If no change has occurred then data that the client already has or an empty string is sent as the response. This is a waste of resources as is the constant querying of the server: creating and closing new connections and sending unneeded messages. A diagram describing the polling technique can be seen in Figure 1. The arrows represent requests from the client and responses from the server.

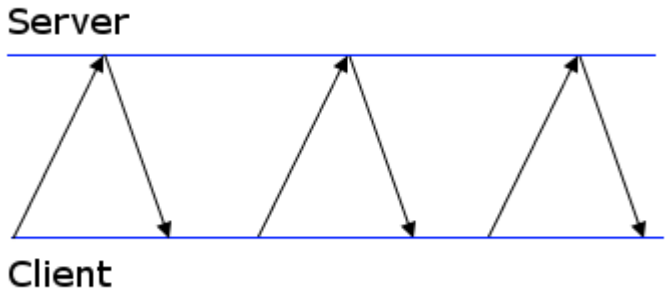


Figure 1 - Polling

A problem with polling is the latency that stems from the very core of the idea: an AJAX script will check for new information only ever so often and any changes that occur between two checks will only be shown once the next check is made. To get lower latency figures, the client would have to set the polling frequency very high, which would mean very high loads on the server and its infrastructure: given a few thousand users each using a high polling frequency, the server's connection would be under quite a load as well [3]. For time-critical information, for example stock prices or auction bids, this latency can cause serious errors.

2.2 Long-polling

An advanced version of polling is called long-polling. Instead of the server sending a response and closing the connection once the client's request is processed, the server doesn't respond anything until new information is available. The connection stays open and only when new information is received will the server send back a response. As the response containing new information is received by the client, a new connection is initiated [1]. This means that there will almost always be an open connection waiting for information and latency is greatly decreased. A simple diagram describing the long-polling technique can be seen in Figure 2.

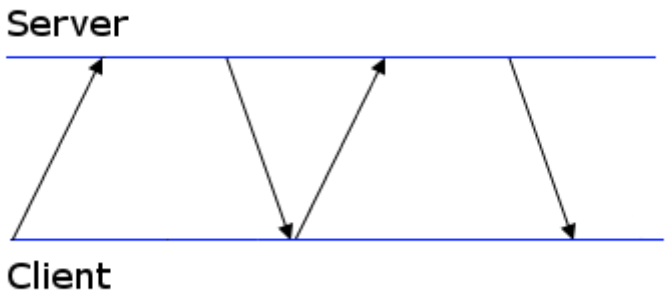


Figure 2 - Long-polling

The downside of this technique is that servers are placed under a greater load, needing to keep up each connection initiated by clients. Latency can still be a factor as well. When two changes occur right after one-another then the first one will be sent to the client. The client will see outdated data (the first change) while the server state has already changed. The next change will be shown once the client has received the previous data, processed it, initiated a new connection and then acquired new information from the server. This can theoretically increase the average latency of this method up to 3 times [21]. If another change occurs during that time then the previous change will go unnoticed by the client. If the updates to information happen frequently then long-polling behaves much like regular polling. If updates are slow, then long-polling resembles streaming [1].

A comparison of this technique to regular polling was done by Engin Bozdag, Ali Mesbah and Arievan Deursen in their article „Performance Testing of Data Delivery Techniques for AJAX Applications"[2].

2.3 Streaming

Another solution is streaming. Instead of the server sending back responses whether it has new information or not, users can subscribe to certain threads and receive information only when it has changed on the server. The server holds the connection open indefinitely and does not require any input from clients after the subscription has been made. This means that data is sent to the client immediately after it has changed, which improves data coherence. No information is sent when it has not changed since the last check-up and new information will be pushed out immediately [2]. Streaming offers great improvement in responsiveness in applications where data coherence is a priority. A diagram describing the streaming technique can be seen in Figure 3.

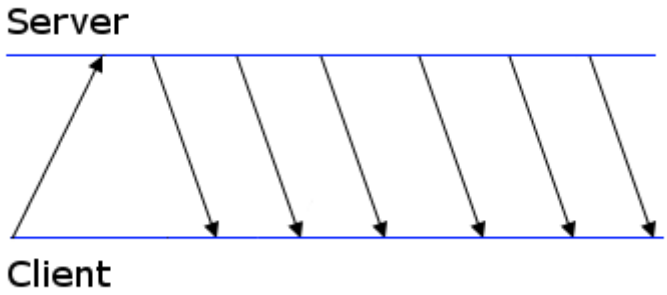


Figure 3 - Streaming

In the current web architecture however, creating streaming applications is not quite as elegant. There are many ways of achieving streaming functionality, but there are significant drawbacks to each, chief among them being browser compatibility.

Cross-browser compatibility has always been a problem when it comes to streaming. In the future, this may be alleviated by the emergence of web sockets or uniform handling of certain AJAX methods.

One solution to this is to focus on a part of the browser market and develop applications that work as intended only on those browsers. This approach is certainly viable for applications developed for use in a network that is not available to the public over the Internet, such as corporate intranets.

2.3.1 XHR

XHR is a JavaScript API capable of connecting to a remote server and transmitting the received data to the user. XHR objects are used for standard AJAX transmissions and also COMET applications. It has the means to store the data and also to report its state. A property called `readyState` is used for the latter. The `readyState` property has five possible values [12]:

0. request not initialized has happened
1. connection established

2. request received
3. processing request
4. request finished and response is ready

Whenever the state changes, even if the change results in the state remaining the same, an `onreadystatechange` event is fired. This allows processing the object accordingly, for example: while the state does not equal 4, show a loading bar to the user.
In streaming applications, `readyState` 3 is the important one. The server will keep a connection open indefinitely using a loop or recursion and the readyState property will never reach 4. When new data is sent to the client, an `onreadystatechange` event is fired and the client can begin manipulating the data to serve it to the user. XHR streaming applications are somewhat similar to forever frames in that JavaScript is used to create an object for brokering the asynchronous transmissions between a server and a client. XHR however is a native JavaScript object, whereas forever frames use a native object of HTML: iframes. While XHR is architecturally simpler to program, the method relies on the data being editable during state 3. This is not possible in any version of Internet Explorer, making this method inadequate for commercial purposes [11].

2.3.2 WebSockets

WebSockets[6] are an HTML5 specification, which includes a new API for the browser and a new protocol for interacting with the server. A WebSocket is an interface for full-duplex single socket connections. They are native to the browser and allow creating powerful applications using only a few lines of simple code. WebSockets allow the browser to initiate and maintain a connection with a server [4]. Unlike traditional HTTP, no headers are sent after the initial connection, which should help use less bandwidth.

Browser support at the time of writing is scarce. At the time of writing, only Google Chrome and Safari have WebSockets turned on by default [5]. Firefox 4 and Opera 11 are able to use WebSockets, but they are disabled by default [10][17].

WebSockets also require a server-side solution to handle the new protocol. Jetty for example, the Java based server engine, has web socket support since version 7.01 [20].

2.3.3 Server Sent Events

Server Sent Events is a web standard that allows transmitting data through a JavaScript object called `EventSource`. Server Sent Events is a standardized version of XHR streaming [8]. Server Side Events do not require special server side support. At the time of writing Safari, Opera and Chrome support Server Sent Events. Firefox does not and neither does Internet Explorer [18].

[6] http://websocket.org/

2.3.4 Forever frame

Neither of the above methods for achieving streaming functionality is currently viable for public consumption. The browser market is divided and while there are no comprehensive browser statistics, the latest sources[7] suggest that 45% of the market share is held by the different versions of Internet Explorer on which XHR streaming will not work and over 70% use older versions of browsers where WebSockets and Server Sent Events are not enabled.

Another technique is the forever frame, which offers cross-browser compatibility. In this case we use a hidden iframe embedded within the web page to connect to the push server. When the server receives the iframe connection, it starts pushing out the required data, but surrounded by `<script></script>` tags, which contain a JavaScript function to be called on the client's side. The data which the client should push out should be the argument of the function. The `<script>` tags are then executed by the browser in order, one after another [7]. While this solution does work similarly across all browsers, it does have drawbacks:

- The server-side code is tied directly to the client-side code, thanks to the need to know the client-side JavaScript function that will eventually output the data.
- When the connection is lost, the iframe itself has no way of restoring it. This means that any number of timeouts, for example the server's default timeout, can break the application. To combat this, additional scripting is required.
- There is no confirmation from the client upon receiving data. This means that the server always assumes that the client received the data and will not try resending it. Any network interference, timeout or a dropped connection can cause data to be lost. This problem is also present within XHR streaming.
- The iframe technique sends data incrementally, but the response in the browser retains all the information received. This means that if the connection is left alive for very long periods of time, the browser would use up a lot of memory to store the data. This constitutes a memory leak.
- Proxy servers, router and other intermediary devices may buffer the output, preventing the client from seeing some or any updates.

For these reasons this thesis strives to create a forever frame plug-in for the jQuery JavaScript framework. The plug-in will alleviate some of the flaws of the forever frame technique in the following ways:

[7] http://gs.statcounter.com/
http://getclicky.com/marketshare/global/web-browsers/

- The call-back function will be standardized. This means that while the server-side implementation still relies on the client-side implementation, specifically it needs to know that the plug-in has been applied; it no longer needs the name of the function. Another possible implementation of this would be to pass on the callback function's name to the server-side script using HTTP's GET method.
- The JavaScript object that creates the iframe will be reset from time to time. This reset time can be configured and it combats the timeouts and memory-leak issues. When a connection times out, it will be restored the next time a reset occurs. Also, the response will be cleaned out and started again, meaning the browser doesn't have to store as much info.

3 Solution

3.1 Description

The newly created application uses the jQuery JavaScript library, a plug-in for said library specifically written for this test and Java as its server-side language. The plug-in, called foreverFrame, will use the forever frame method to request data from the server. The application will be published as a JSP page.

The push servlet uses classes implemented in other CHIRON applications, namely `StockData` and `PublishService`. The latter acts as a publish server while the former is the object representing data sent to the clients. As the publish server is realized as a Java class and needs to be manually started, this application is only usable within the testing framework. For a similar live application, a stand-alone publish server would be required.

The servlet has an internal ID counter and a nested `Client` class for storing client info. GET requests are expected to come from clients connecting to the application. POST requests are expected to contain new information that is to be distributed to the clients. When each client connects, they send a GET request with the `init` parameter set to `getId`. A new `Client` object is created, given a unique ID and placed in a vector of clients. The ID is given from a counter that is incremented each time a new connection arrives. Once the client has received an ID, it starts a new connection using the ID as a GET parameter. The request is then placed in a suspended state until new information is sent via a POST request.
POST requests are expected to come from the publish server. Each POST request contains stock information for all 3 stocks, the number of the published message and the interval at which the messages are sent.
The date and time when the data was sent is added to calculate latency during testing. A variable that tells whether there are more messages incoming or not is also included. This info is assigned to class variables. The exact names and types of these variables can be seen in Figure 6.

Once a new POST request comes, the vector of clients is iterated over and each client is brought back from a suspended state. This process is repeated for each test, until the `end` variable is set to `true`. At that point, the connection will be closed. Figure 4 shows these steps in a sequence diagram, where the service provider sends 2 messages.

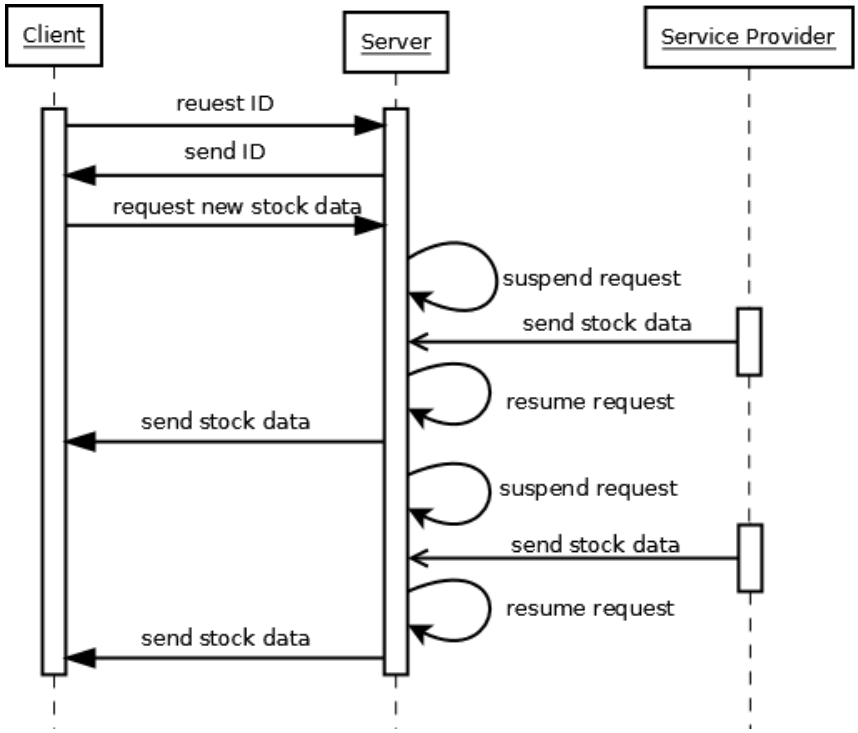


Figure 4 – Sequence diagram of the streaming application sending 2 messages

3.2 Realization

As the test is run on Jetty, continuations are used. Continuations are a Jetty implementation of suspendable requests [19]. Suspendable requests will become a standard in the servlet specification 3.0, at which point the Jetty implementation will be replaced. These allow Jetty to free up a thread when the thread is not used, allowing for greater scalability. The thread will be resumed when a timeout occurs or when the resume method is called on the object. Creating a continuation can be done by:

```
Continuation continuation = ContinuationSupport.getContinuation(request);
```

Note that Continuation and ContinuationSupport classes must be included. This declaration creates a continuation and ties it to a specific request. The continuation can be suspended by using:

```
continuation.suspend(timeout);
```

Timeout is represented in milliseconds and must be a numerical value. The continuation is resumed when the timeout occurs, or when a resume() method is called. If timeout is set to 0 then the continuation will remain suspended until resume() is called.

An important part of COMET techniques is flushing the buffers, as the connection is not really closed and the client needs to receive the data somehow. This can be achieved by using:

```
response.getWriter().flush();
```

Figure 5 shows the class diagram for the completed application. The `doGet()` method is used both for sending out the ID and the stock data. `doPush()` is the recursive method that sends out data. `doPost()` is where the ServiceProvider sends data.

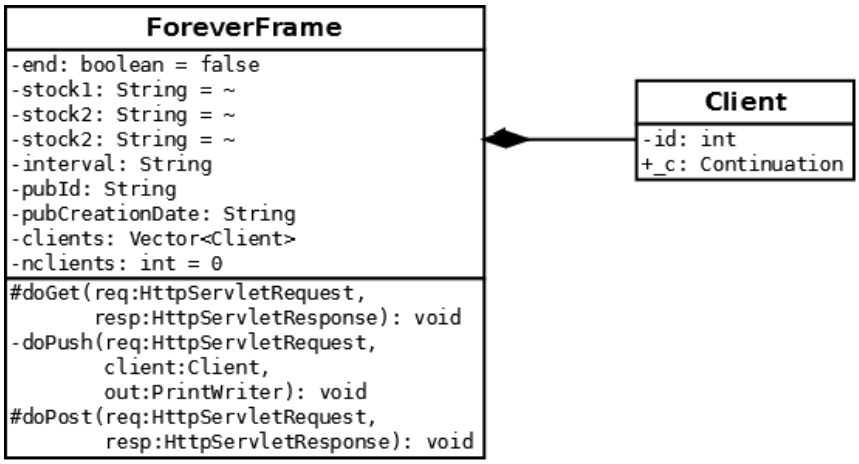


Figure 5 – Class diagram of the completed application

Note that the application is intended to be used only in conjunction with CHIRON and its tests. There are no checks made to determine where he data published to the users is coming from, so if it were used in a public space, anyone could make a POST request and send falsified data to the clients.

The plug-in, called foreverFrame, which was developed to go along with the test application, can be used in any project requiring COMET through the use of forever frames. ForeverFrame allows for two different modes of operation. The first is if it is needed to simply update a single element on the page with new information. To initiate the plug-in in such a situation, one would write

```
$(function() {
	foreverFrame.init({
		src: 'http://example.com',
		target: 'targetId'
	});
});
```

The plug-in will then create a new hidden iframe when the DOM of the document was fully loaded. The src parameter is the server address where the iframe will be pointed to. The target parameter tells the plug-in where to insert any data received from the server. The data can also be shown in different locations using the optional targetType parameter with the value „.“ (default is „#“ or the id selector), which is the class selector in jQuery. The desired targets should therefore have the same class.

The second situation, one that is encountered in the test application as well, is when the data from the server should be distributed across multiple locations. This can be achieved by writing

```
$(function() {
	foreverFrame.init({
		src: 'http://example.com',
		target: 'example1,example2,example3',
		multipleTargets: true
	});
});
```

The multipleTargets parameter tells the plug-in that the data should be distributed. The targets are separated by a comma and passed along as HTTP's GET method parameters to the server in the form of target1=example1, target2=example2 and so on. Multipart=true and the number of targets is also sent to aid in handling the target variables.

As this is a client-side plug-in, both situations require server-side implementations as well. In the first case the server-side code simply needs to call out the execute function of the iframe's parent element and pass any data it has to send as the function's variable. The string it needs to output should look something like this (in Java):

```
"<script type='text/JavaScript'>parent.foreverFrame.execute("+data+");</script>"
```

In the second situation the demands on the server code are more complex. The server has to create a JSON object of the data it has and pass it along as the argument of the execute function. The argument parameters are the targets where each part of the data should be distributed to. These can be hard-coded into the server code, but for convenience, they are also added at the end of the server URL as GET variables. Using the example above, in this case the data variable should look like this:

```
"{example1:"+data1+", example2:"+data2+", example3:"+data3+"}"
```

Another optional parameter is timeout, which sets the interval at the end of which the forever frame will be reset. The latter, if set to true, will alert the user if their browser does not support iframes.

Of note in the architecture in both the newly created application and the previous ones found in the CHIRON framework is that the stock values are published as a single message from the server. This means that when one of the stock prices changes, all stock prices will be sent by the server in the message instead of just the one. While this is an improvement over the polling technique, it still means that there is data redundancy. The alternative is to create a new subscription for each stock price however, and keeping three (in the case of the test applications) connections open is wasteful of resources.

4 Experiments

The steps that will be undertaken by the author are outlined in this section.

1. Create a plug-in for the JavaScript library jQuery that would ease the development of COMET applications using the forever frame technique.
2. Use the plug-in created in step 1 to create a push web application similar in functionality to the ones proposed by E. Bozdag et al.
3. Create a testing suite for Chiron that would allow testing of applications using the forever frame technique. Creating said suite requires:
    a. The web application created in step 2.
    b. A .py script for grinder to handle the specific test of the web application
    c. A new test in the ChironController's main method.
4. Using the AJAX testing platform CHIRON, repeat the tests run by E. Bodzdag et al for comparison purposes. Then run the same test on the application created in step 2.

The client simulator, statistics server and the Chiron controller will run on a desktop computer with a Core 2 Duo Intel processor running at 3GHz. The computer has 8GB of DDR2 RAM and is running Ubuntu 10.04 as its operating system. The application server will be run on a laptop computer with a Core 2 Duo Intel processor running at 2.4GHz. The laptop computer has 2GB of DDR2 RAM and is also running Ubuntu 10.04 as its operating system. The computers will be linked using a dedicated Gigabit network provided by a Linksys (model no.) router.

The application server will be run on a different computer to monitor the impact of an increasing number of users. This way no other processes running on the server will interfere with the load measurement results.

Each test will be run 10 times.

4 different applications will be tested: polling, Bayeux, DWR and streaming. The polling application will use the polling method with different polling frequencies: 1, 5, 15, 30 and 50 seconds. The Bayeux and DWR applications are both variants of the long-polling method. These applications are all included with the CHIRON framework [2]. The fourth is a streaming application developed by the author, the structure and contents of which is detailed in section 3.2.

The intervals at which new data is published are 1, 5, 15, 30 and 50 seconds.

4.1 CHIRON

CHIRON[8] is a load testing framework for asynchronous internet applications [2]. Version 1.2 is used in this thesis. The framework allows running tests on multiple computers while coordinating the whole test from a single machine. The requirements for running CHIRON are a UNIX system (or a capable emulator) and Java SDK version 1.5 or greater. CHIRON is written in Java.

CHIRON consists of 5 parts: a statistics server, a client simulator, an application server, a service provider and the controller. The controller is a Java application that coordinates all the other services. The statistics server is a Log4j[9] SocketServer, which receives logging info from all the other tools. The client simulator is The Grinder[10], a Java load testing framework. It is equipped with the necessary Jython scripts to access and manipulate any info that is sent to the clients by the application server. The application server is the web server serving up all the test applications. Jetty 6 is used as the web server. The service provider is a Java program that sends random stock data as well as test specific info to the application server at the configured intervals. All of these parts can be spread over multiple computers. While it is not necessary, it is at least recommended that the application server and the client simulator run on different computers as the load from one can interfere with the performance of the other, thereby skewing the experiment results.

Setting variable values such as polling and publishing frequency (see Section 4.2) and general test setup is done through experiment.properties file. Changing this file is all that is needed to rerun any configured test with new variables or in a new setting.

Once an experiment has been started, CHIRON will start looping through all the tests. In order, they are streaming, Bayeux, DWR and pull. Each test will loop through all the active connection and publish frequency settings and record results. Pull will also loop through the different polling interval settings.

Figure 6 shows all the steps taken in every iteration of the experiment.

[8] http://spci.st.ewi.tudelft.nl/chiron/
[9] http://logging.apache.org/log4j/
[10] http://grinder.sourceforge.net/

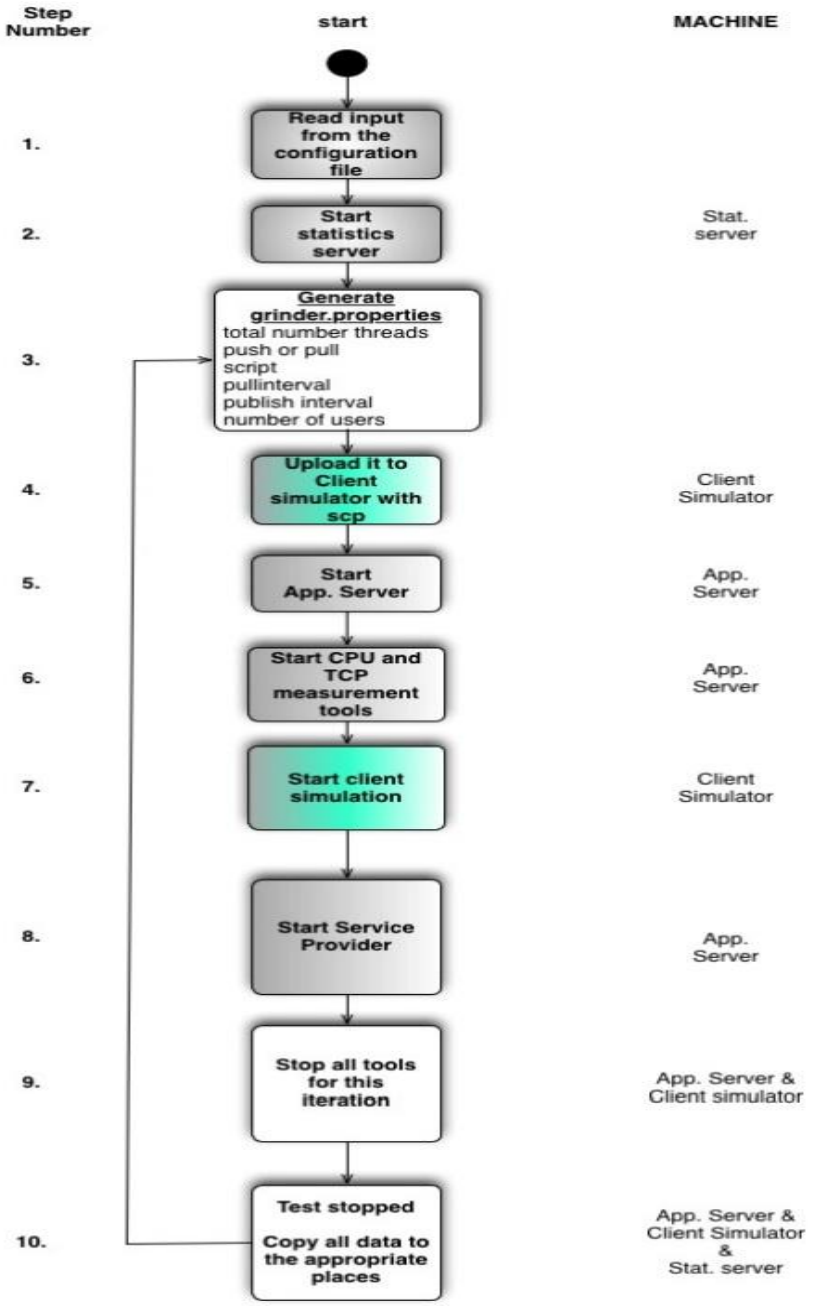


Figure 6 – the steps CHIRON goes through each iteration [2]

CHIRON uses different native UNIX tools to measure and log data. CPU usage is measured using top[11], a tool designed to measure CPU usage in real time. Top is set to take snapshots of the CPU usage on the application server each second and save them for later editing. TCPdump[12], a UNIX packet sniffing tool, is used to capture TCP packets coming from the application server or the client simulator.

[11] http://www.unixtop.org/
[12] http://www.tcpdump.org/

4.2 Experimental Settings

4.2.1 Load on the application server

Load on the application server will be measured with the Linux tool top, which shows all the applications using the CPU in real time. Top will measure the application server's java process and capture its status every second. Measuring load is important in determining whether a test application is scalable [2].

4.2.2 Messages received by the client

Each message received by the client will be noted. This is important to see if there are any dropped messages or if there are too many messages sent by the server. The correct number of messages received should be 10, as the tests will be run 10 times. If there are too many messages, then there will be data redundancy: messages that the client already has are resent creating needless load on the application server. If however there are less than 10 messages, then that means some of the messages have been dropped and the client has not received that information, ie. there are problems with data coherency [2]. This will also be checked by counting the number of unique messages received. Due to streaming limitations, no error control could be implemented so the server always assumes that the data reaches its destination. This is not always the case, as a large number of factors, including timeouts, dropped connections, lag or buffering from intermediary devices and so on, can cause the message to be dropped. This means that the client might never see certain points of data, which is a big problem in cases where data coherence is important.

4.2.3 Unique messages received by the client

From all the messages received by the client simulator, unique ones will be filtered out and counted for each application. This is to make sure that all messages sent by the service provider reach the clients. The number should be 10, as that is the number of messages being sent. Unique messages will be counted in addition to all the messages, because even if the number of all messages exceeds 10, it is not guaranteed that some were sent multiple times and some not at all. This applies to the pull application [2].

4.2.4 Round trip time

Round trip time is the time it takes for the message to get to the client after it's been published [2]. The greater the round trip time, the longer it takes for the message to reach the client. The longer the message takes to reach the client, the greater is the chance that another change already occurred and the client is seeing outdated data. As the streaming application uses only a single connection for the entire test, whereas the rest use at least a connection per published message, the end point of the connection cannot be used to calculate round trip times in this instance. Instead, the point in time at which a packet containing a message is received by the computer simulating active connections is

used. This gives an advantage to the streaming application. To rectify this only the last published message in each test is measured. This gives a much lower data set, but uses the same measuring method as do all the other tests and is therefore comparable.

4.2.5 Network traffic

All network traffic within the network will be monitored. Packets are captured using tcpdump, a Linux packet sniffing tool. Only packets moving between the application server and the client simulator will be used to filter out any errant packages [2]. The fewer packets are sent within the network during testing, the less congested the network will be, leaving resources for other tasks. If the network is flooded with too many packages, then other services running on the same network might suffer, as well as the application itself.

4.2.6 Publish interval

Publish interval is the interval at which new messages are sent to the server by the publishing agent [2]. This can be anything from another end user to a different server. In the test application this will be simulated by the CHIRON load testing framework. While in real applications the publish interval is usually random, in the test application fixed publish intervals will be used. This is to guarantee a fair comparison between different AJAX solutions. The publish intervals are set at 1, 5, 15, 30 and 50 seconds. The variance helps to see how the long-polling and streaming applications perform when messages are sent often, meaning frequent reconnects for long-polling and when messages are sent further apart from each other, meaning long periods of keeping the connections open.

4.2.7 Active connections

Active connections are clients that are connected to the application server. The number of connected clients will be measured to see how different solutions scale under increasingly difficult circumstances [2]. The Linux tool TCPdump, which monitors an active network interface, will be used to monitor connections to and from the application server. As the computer used in these tests is not as powerful as the supercomputer used by E. Bozdag et al, the number of active connections for each test will be quotients of the numbers of active connections used in their article. During testing, active connections over 100 caused a dramatic drop in the performance of all tests. As this could be not attributed to poor performance on the side of the application server, it was concluded that the problem lay with the computer simulating the connections. Due to this limitation 10, 50 and 100 active connections will be used during the tests.

4.2.8 Poll interval

Poll interval only applies to the polling solution. Poll interval is the interval at which the client asks the server for new information [2]. Different polling intervals will be measured to see which offers the greatest gain/loss ratio in terms of data coherence and sparing resources in the test application. The polling intervals are set at 1, 5, 15, 30 and 50 seconds.

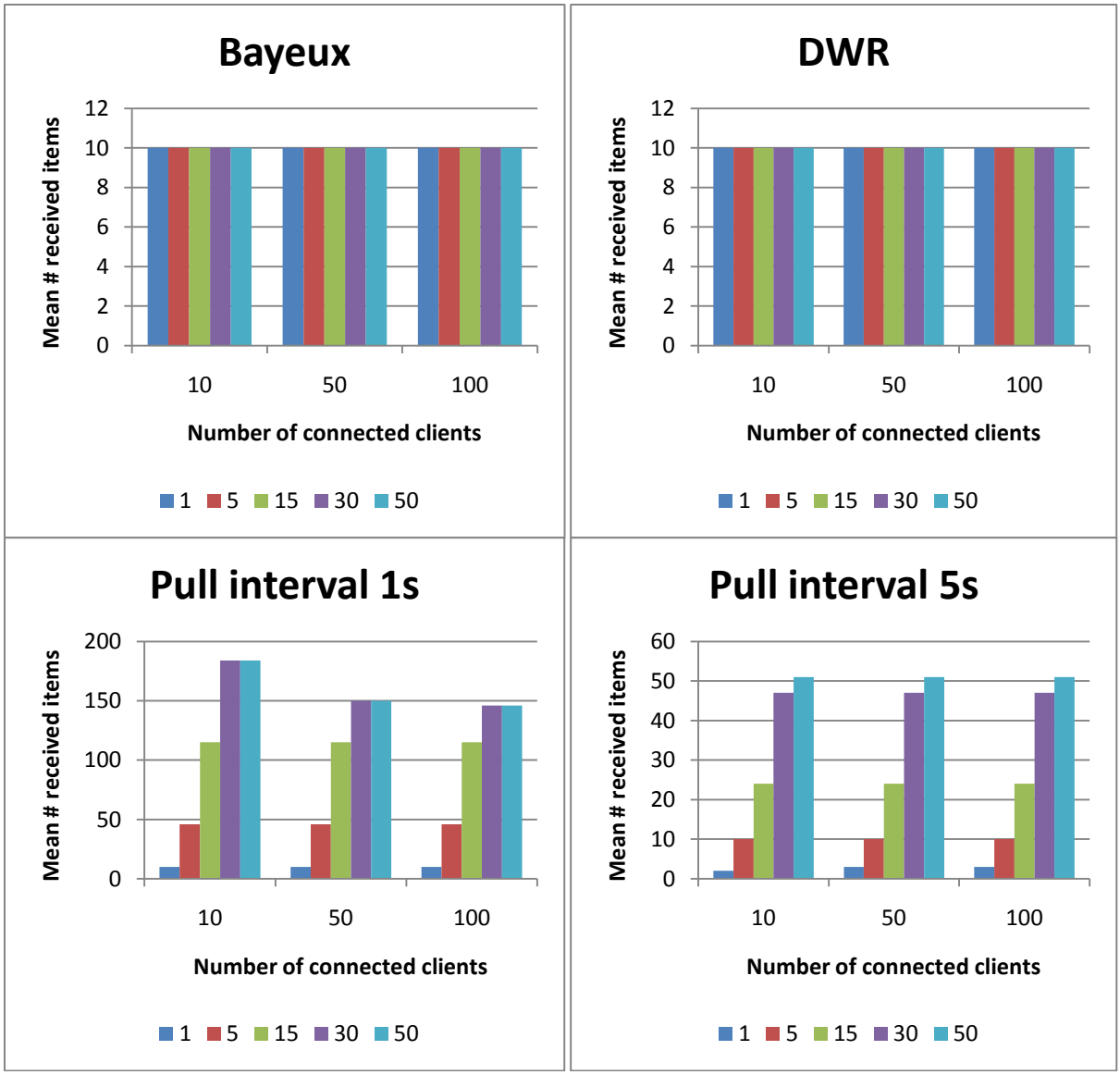
4.3 Results

4.3.1 Number of messages

Figure 6 shows the mean number of received messages by the client for all tests. As predicted, polling resulted in a much greater amount of messages when the polling frequency was low and much lower amount of messages when the polling frequency was high when compared to Bayeux, DWR and streaming versions. When using a 1 second polling interval, the number of messages received reached 184. With 100 users connected, the client simulator became unstable and was unable to process the received messages. This explains the sudden drop seen in the graph. When the polling frequency was decreased to 5 seconds, the number of messages received reduced to a maximum of 51. While the publish interval was set to 1 second, polling frequency of 5 seconds was unable to obtain all the messages. 3 messages were obtained. This means that there was data loss of up to 70% while the polling frequency was 5 seconds and up to 174 redundant messages when the polling frequency was 1 second. When the polling frequency was decreased to 30 and 50 seconds, only 2-4 messages were delivered on average. This was seen in repeated tests, but the exact cause was hard to debug.

Both Bayeux and DWR preformed well, as both always received 10 out of 10 messages.

Streaming performed almost as well as the long-polling solutions, producing 10 messages on average. The average number of messages received was 8 when the number of active connections was 10 and the publish interval was 30 seconds and 9 when the number of active connections was 50 and the publish interval was 1 second or 50 seconds. Scaling the number of active connections to 100 did not produce any problems. Figure 7 shows that not all streaming messages reached their destination. As these messages are not resent due to the nature of streaming, this will cause problems in applications where data coherence is important (see Section 4.2.2). If that is the case then it would be better to use long-polling, as errors in data coherency can be checked there by checking the ID or date of the last message received and comparing it to the next one sent by the server.


Bayeux
Mean # received items
12
10
8
6
4
2
0
10
50
100
Number of connected clients
1 5 15 30 50
DWR
Mean # received items
12
10
8
6
4
2
0
10
50
100
Number of connected clients
1 5 15 30 50
Pull interval 1s
Mean # received items
200
150
100
50
0
10
50
100
Number of connected clients
1 5 15 30 50
Pull interval 5s
Mean # received items
60
50
40
30
20
10
0
10
50
100
Number of connected clients
1 5 15 30 50

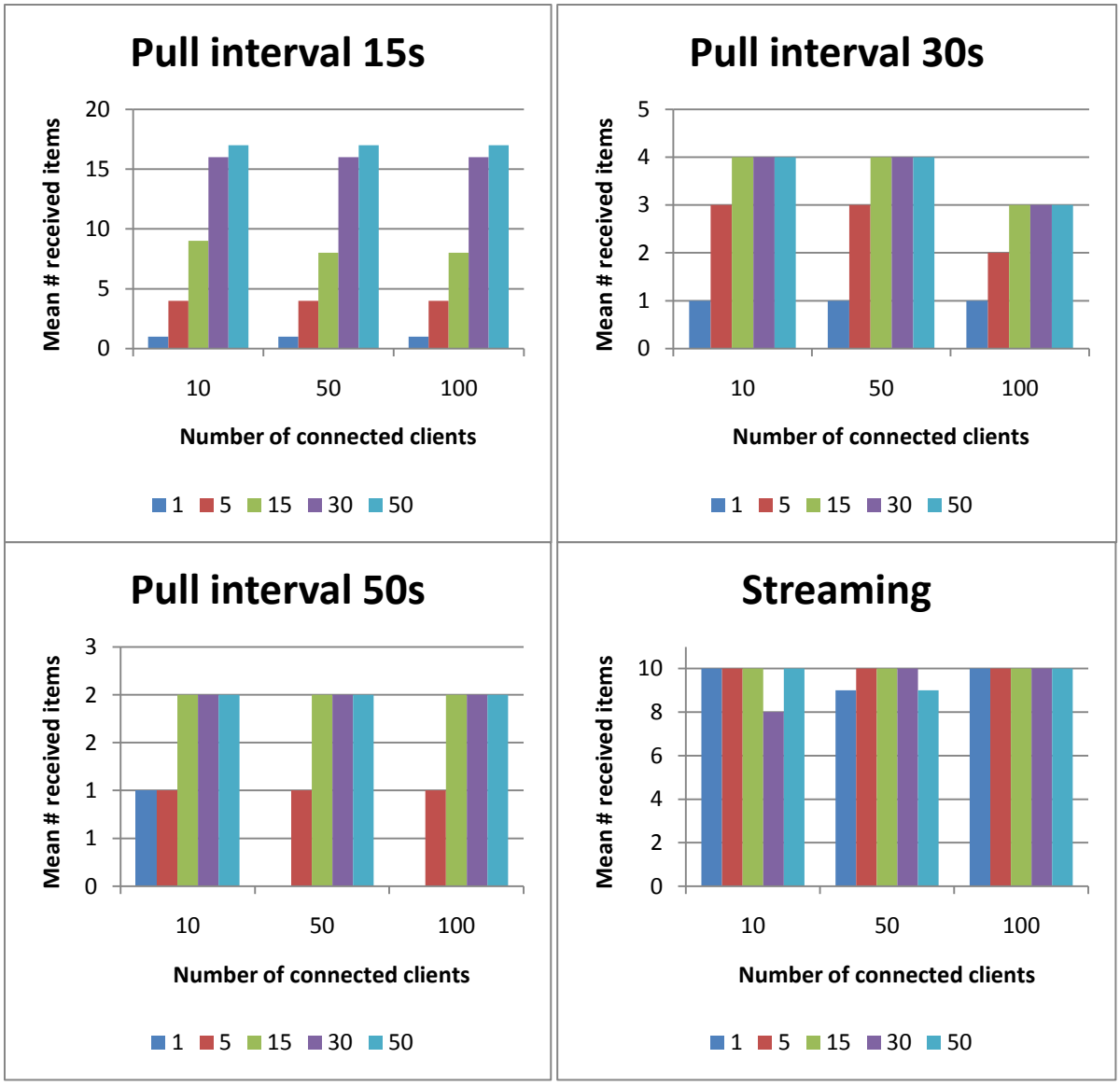


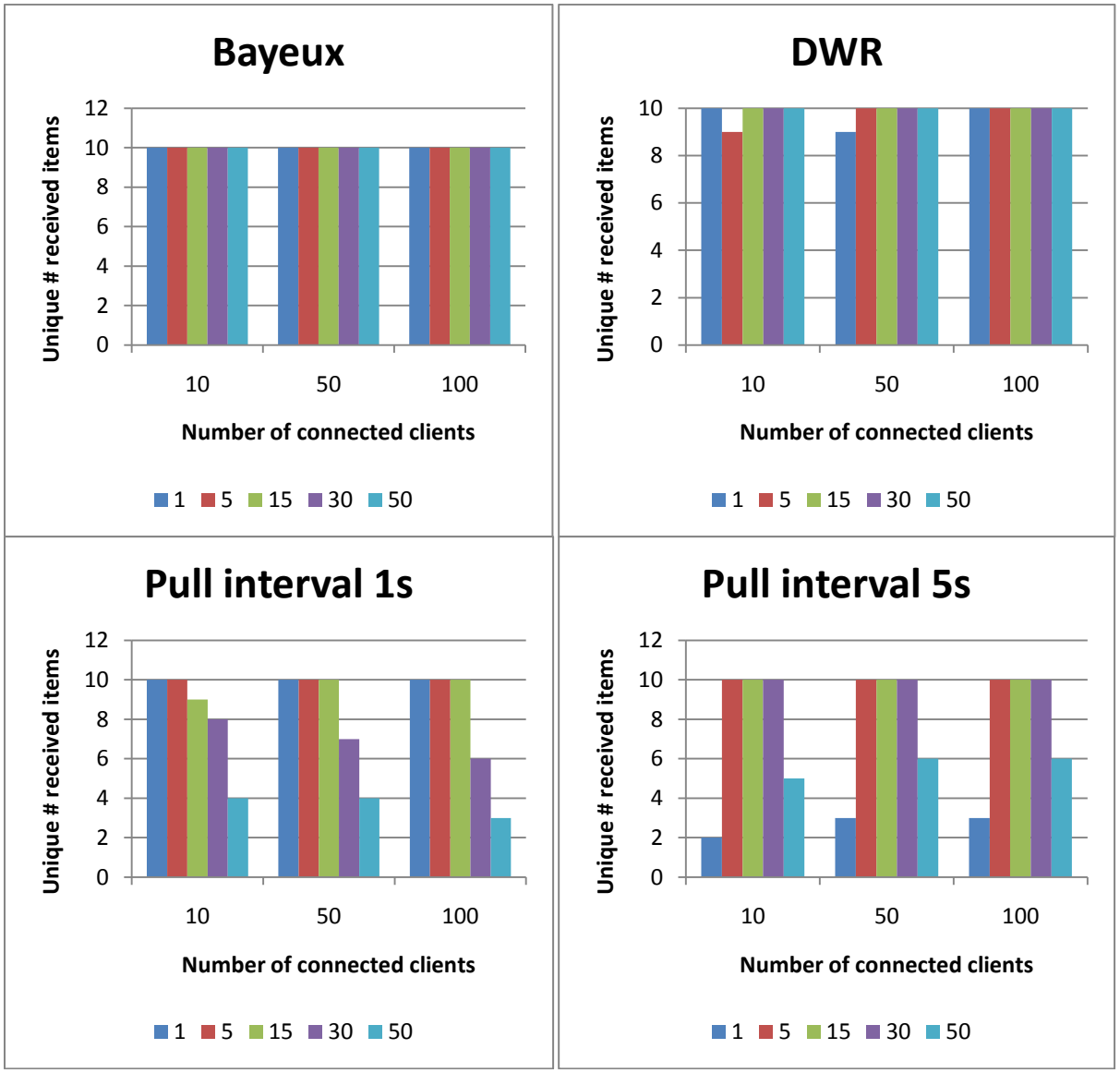
Figure 7 – Mean number of messages received for all tests

4.3.2 Number of unique messages received

Figure 8 shows unique messages that were received by the clients. Numbers for the long-polling and streaming versions match the number of total messages these applications received. The architecture of those applications means that no message is sent twice. This does not waste network resources, but can also cause problems when the messages are not received by the clients (see Section 4.2.2).

The pull application performed reasonably well while the polling frequency was high. At a 1 second polling frequency most messages were delivered. At 1, 5 and 15 second publish intervals, all messages were delivered, the only exception being when the number of active connections was 10. When the frequency was lowered to 5 seconds, the 1 second publish interval reduced drastically in data coherence. An average of 3 messages was received. However more messages were received with a

higher publish interval. This trend continued when the polling frequency was lowered further: the bigger the publish interval, the more messages were received.



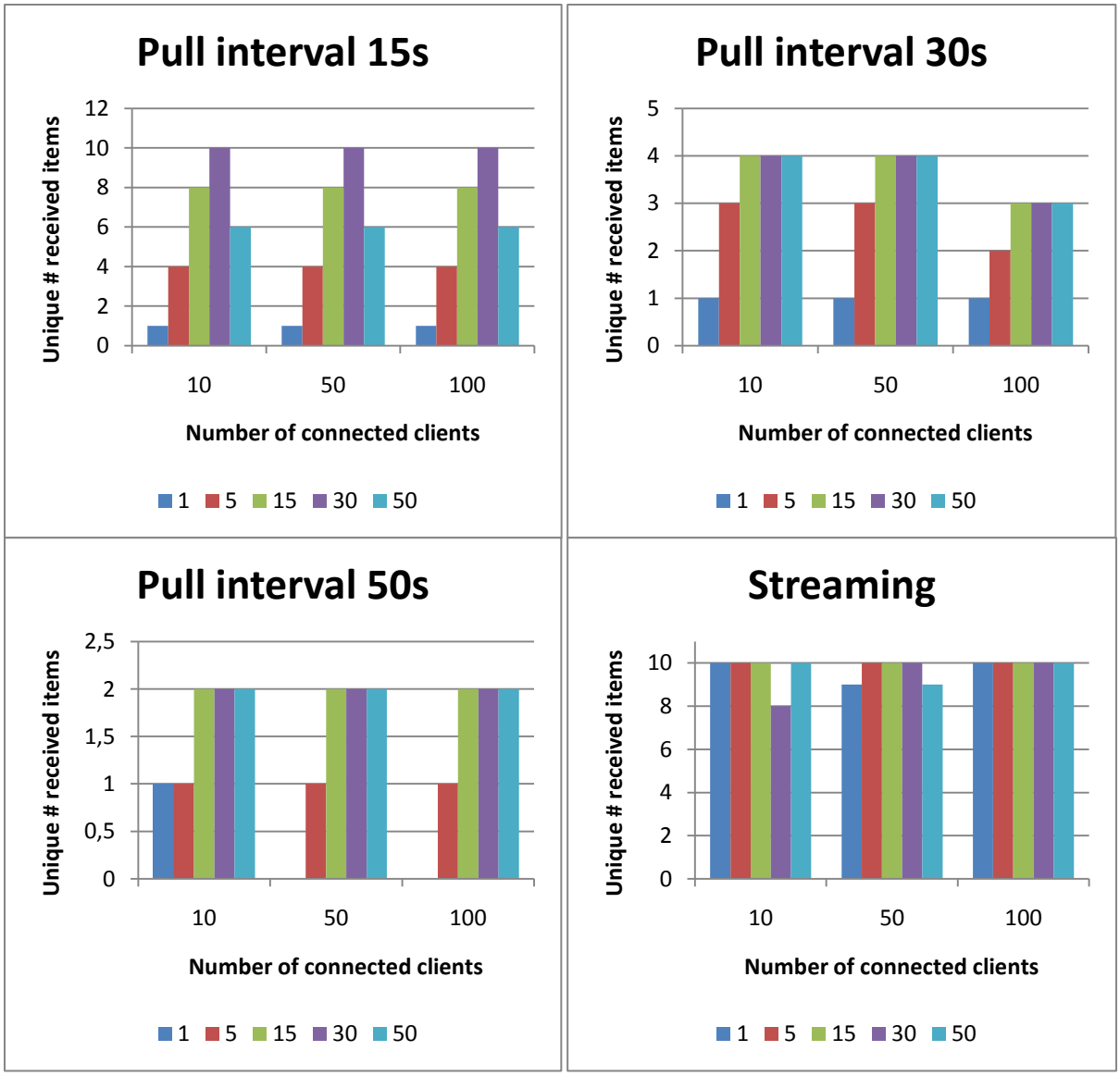


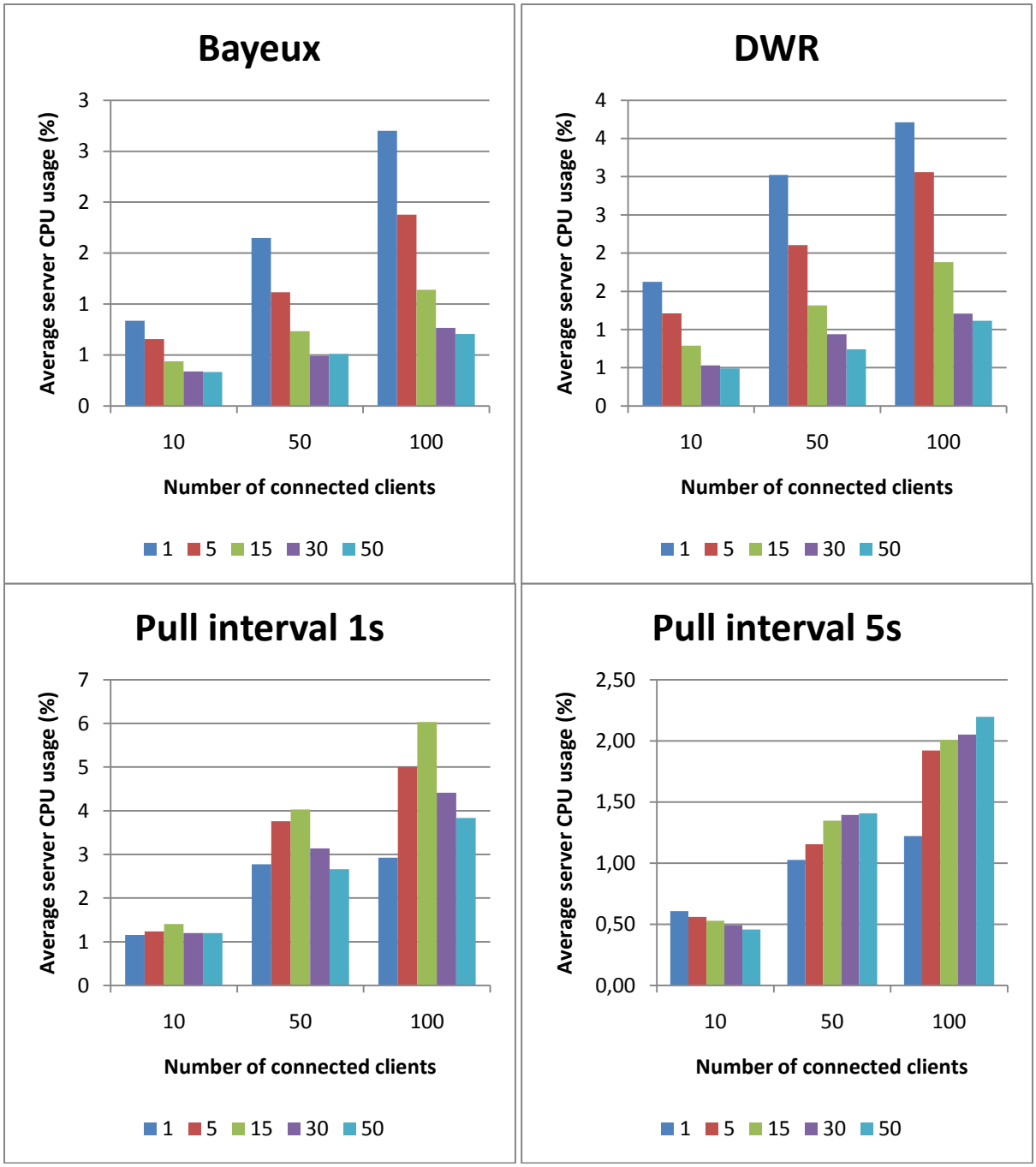
Figure 8 – Unique number of received messages

4.3.3 CPU load

Figure 9 shows the CPU usage time for all the tests. Polling shows a tendency to take up more server CPU time the shorter the polling frequency is. At 1 second the average server CPU time reached a maximum of 6%. This dropped to a maximum of 2,2% when the polling frequency was decreased to 5 seconds. Further decreases to the polling frequency seemed to have very little effect, as the numbers remained nearly identical from intervals of 15 seconds to 50 seconds. The more users were connected the higher the CPU usage was.

Bayeux and DWR long-polling solutions use a similar amount of CPU time as polling solutions when the number of users is low. As more users are connected, the amount of CPU time used increases at a much greater rate, reaching their respective maximums at 3% for Bayeux and 3,7% for DWR. Both used much more CPU time when the publish frequency was low, meaning that holding a connection open was not as expensive in terms of performance.

Streaming behaves similarly to the rest when the number of connected clients is low. As more clients connect, the amount of CPU time used on the server increases significantly. The exact reason for this was difficult to debug as the behaviour of the application was unpredictable in terms of CPU load, but it is the opinion of the author that the heavy CPU usage can be attributed to the basic nature of the streaming application and the handling of continuations by Jetty. Of note is that the server is using more CPU time the longer the publish time is and hence, the longer it takes to keep connections open.


Bayeux
Average server CPU usage (%)
0
1
1
2
2
3
3
10
50
100
Number of connected clients
1
5
15
30
50
DWR
Average server CPU usage (%)
0
1
1
2
2
3
3
4
4
10
50
100
Number of connected clients
1
5
15
30
50
Pull interval 1s
Average server CPU usage (%)
0
1
2
3
4
5
6
7
10
50
100
Number of connected clients
1
5
15
30
50
Pull interval 5s
Average server CPU usage (%)
0,00
0,50
1,00
1,50
2,00
2,50
10
50
100
Number of connected clients
1
5
15
30
50

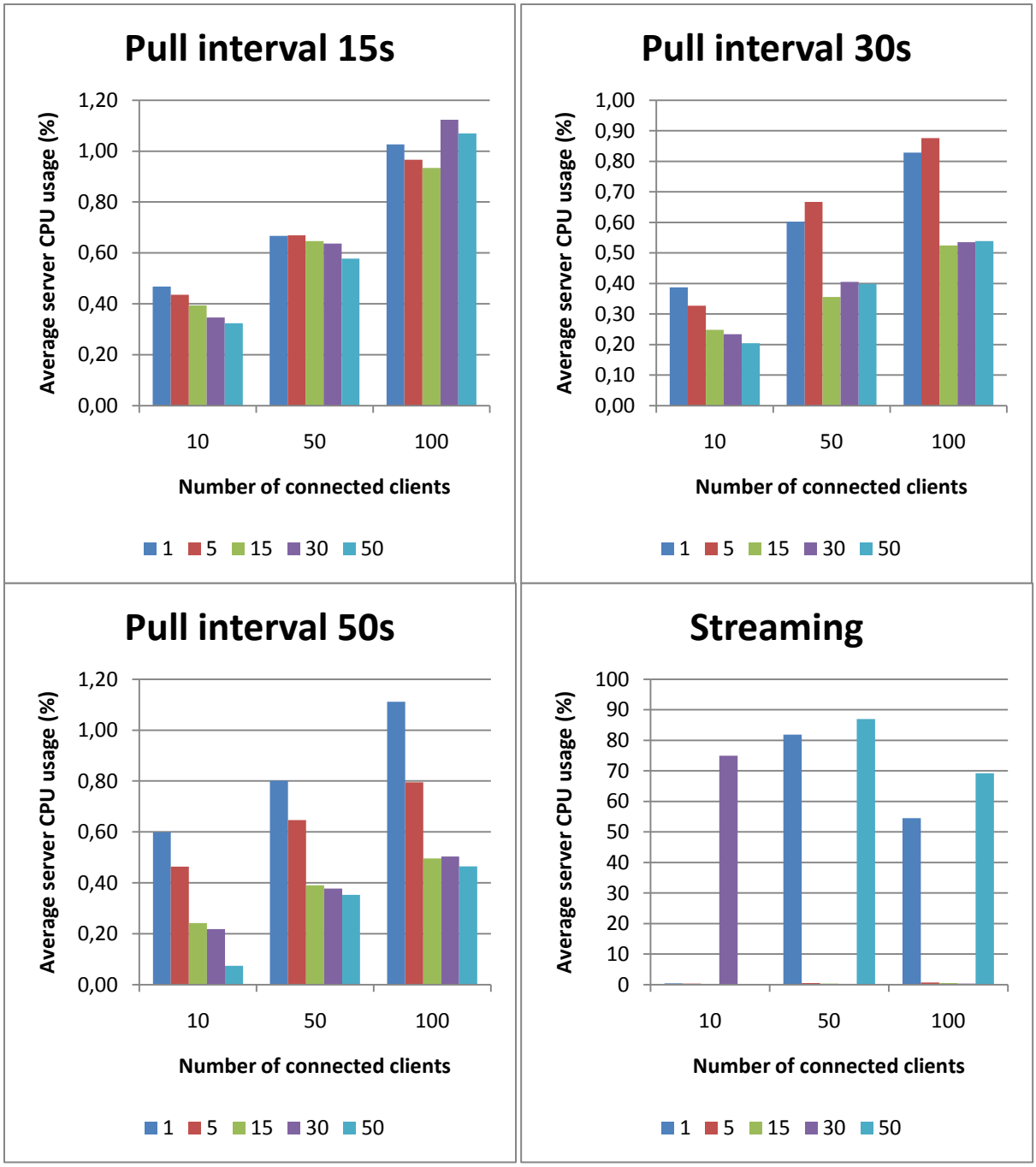
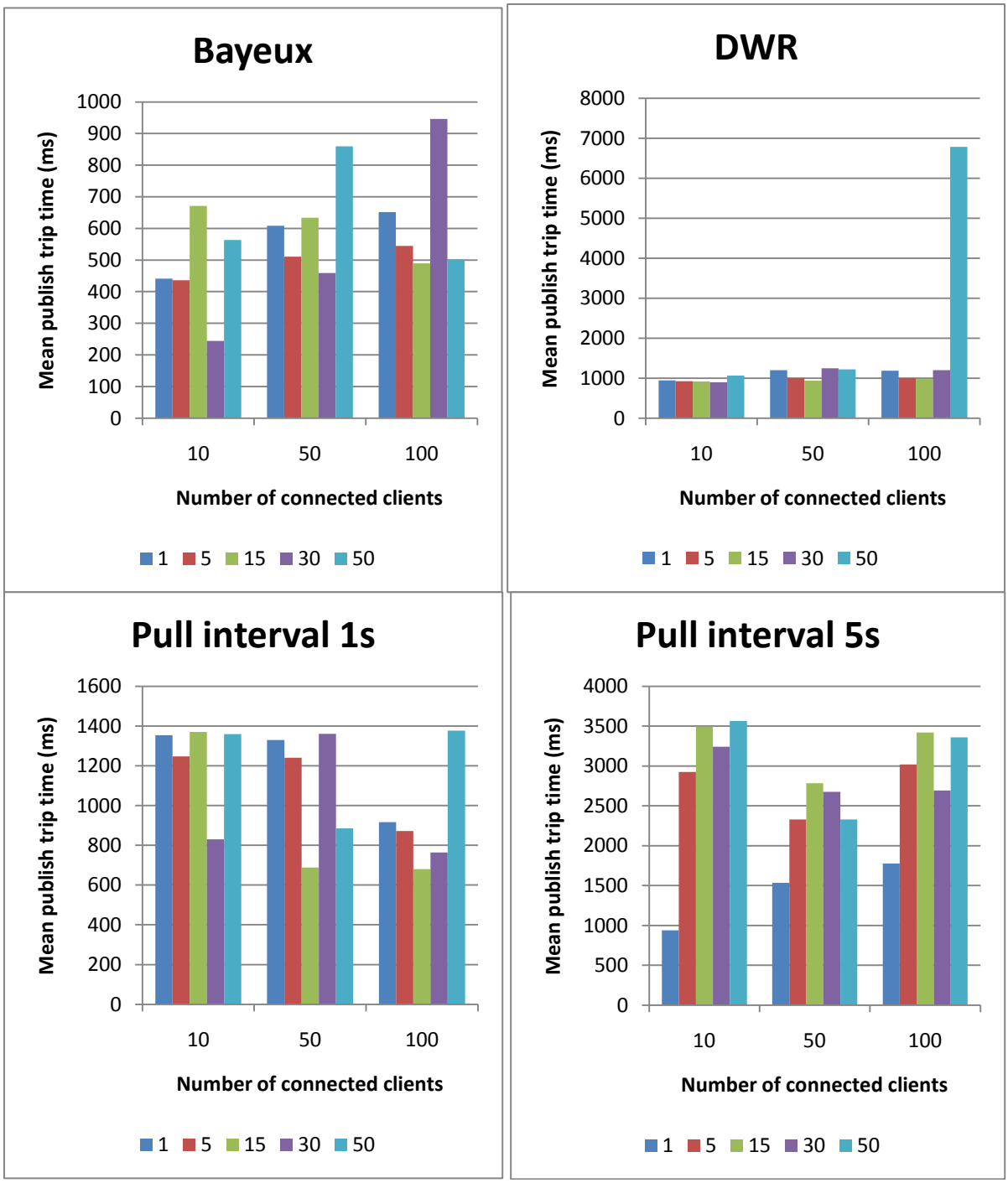


Figure 9 – Mean server CPU time usage for all tests

4.3.4 Trip times

Figure 10 shows the average time it took for a message to reach the client from the moment it was published. Bayeux long-polling method has the lowest latency. The maximum average latency was 946ms and the minimum average latency was 243ms. Bayeux was followed closely by the streaming and DWR versions. While streaming did come close to Bayeux a few times, it performed worse in most cases. The lowest latency was 505ms and the average latency 1305ms. Streaming reached a maximum latency of 2005ms. DWR was fairly consistent, staying between 900ms and 1240ms the entire time, until the number of active connections was increased to 100. Then the average latency reached a maximum of 6783ms. The theoretical latency of a message being sent via the long-polling method can be up to 3 times greater than that of the streaming method, but real-world tests do not mirror this theory. Trip times were very similar between long-polling and streaming and in many cases, long-polling had lower latency times. While this can be attributed to the very basic nature of the streaming application, the theoretical gains in latency achieved by optimizing the application should not provide significantly lower latency times than those of the long-polling version. Latency times do not seem to justify using streaming, given that streaming has significant drawbacks in other areas (see Section 2.3.3).

Pull applications did not perform as well. The trip times were high, up to 18200ms. This does not take into account the fact that very little messages were actually delivered to the client.


Bayeux
Mean publish trip time (ms)
1000
900
800
700
600
500
400
300
200
100
0
10
50
100
Number of connected clients
1
5
15
30
50
DWR
Mean publish trip time (ms)
8000
7000
6000
5000
4000
3000
2000
1000
0
10
50
100
Number of connected clients
1
5
15
30
50
Pull interval 1s
Mean publish trip time (ms)
1600
1400
1200
1000
800
600
400
200
0
10
50
100
Number of connected clients
1
5
15
30
50
Pull interval 5s
Mean publish trip time (ms)
4000
3500
3000
2500
2000
1500
1000
500
0
10
50
100
Number of connected clients
1
5
15
30
50

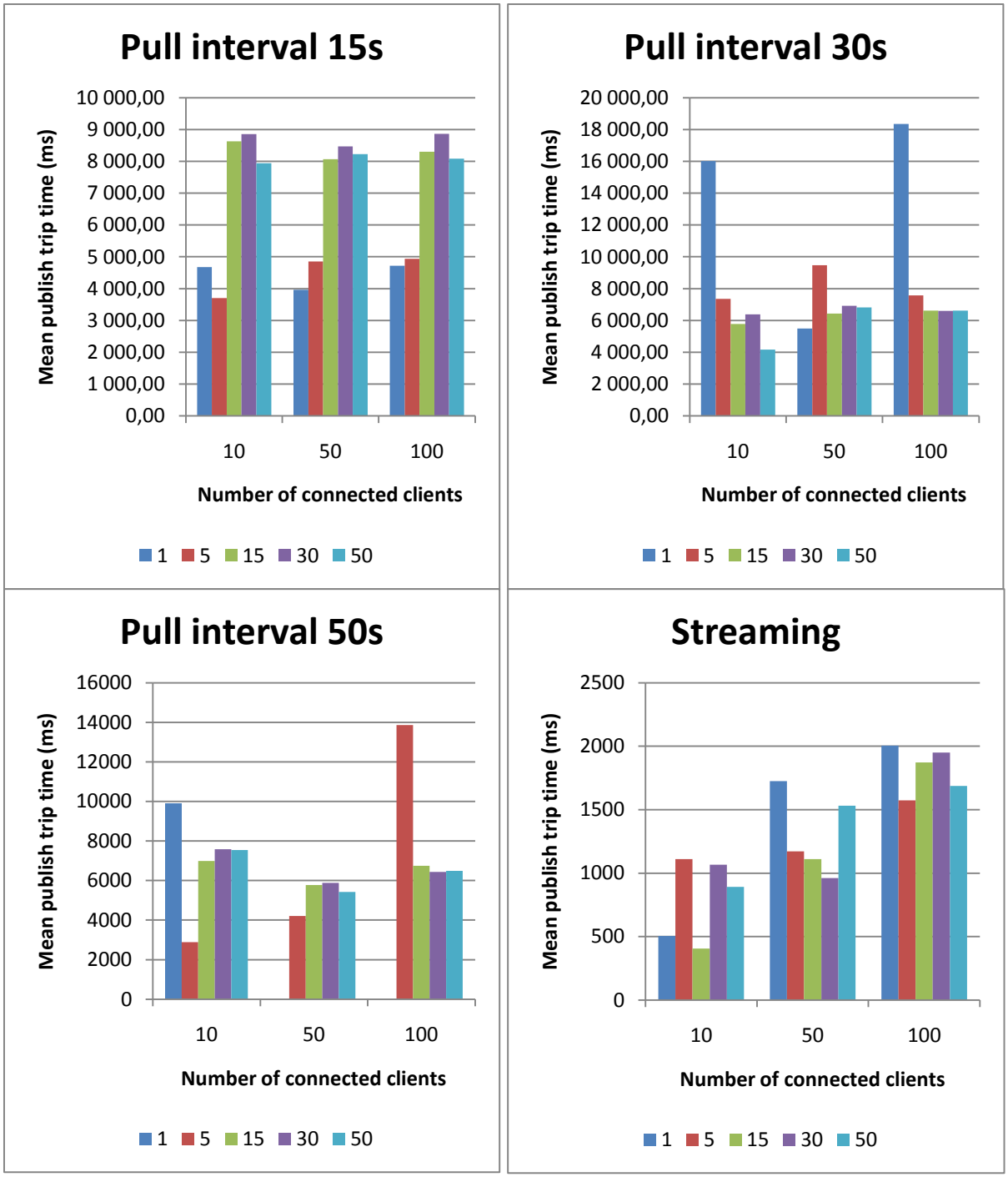
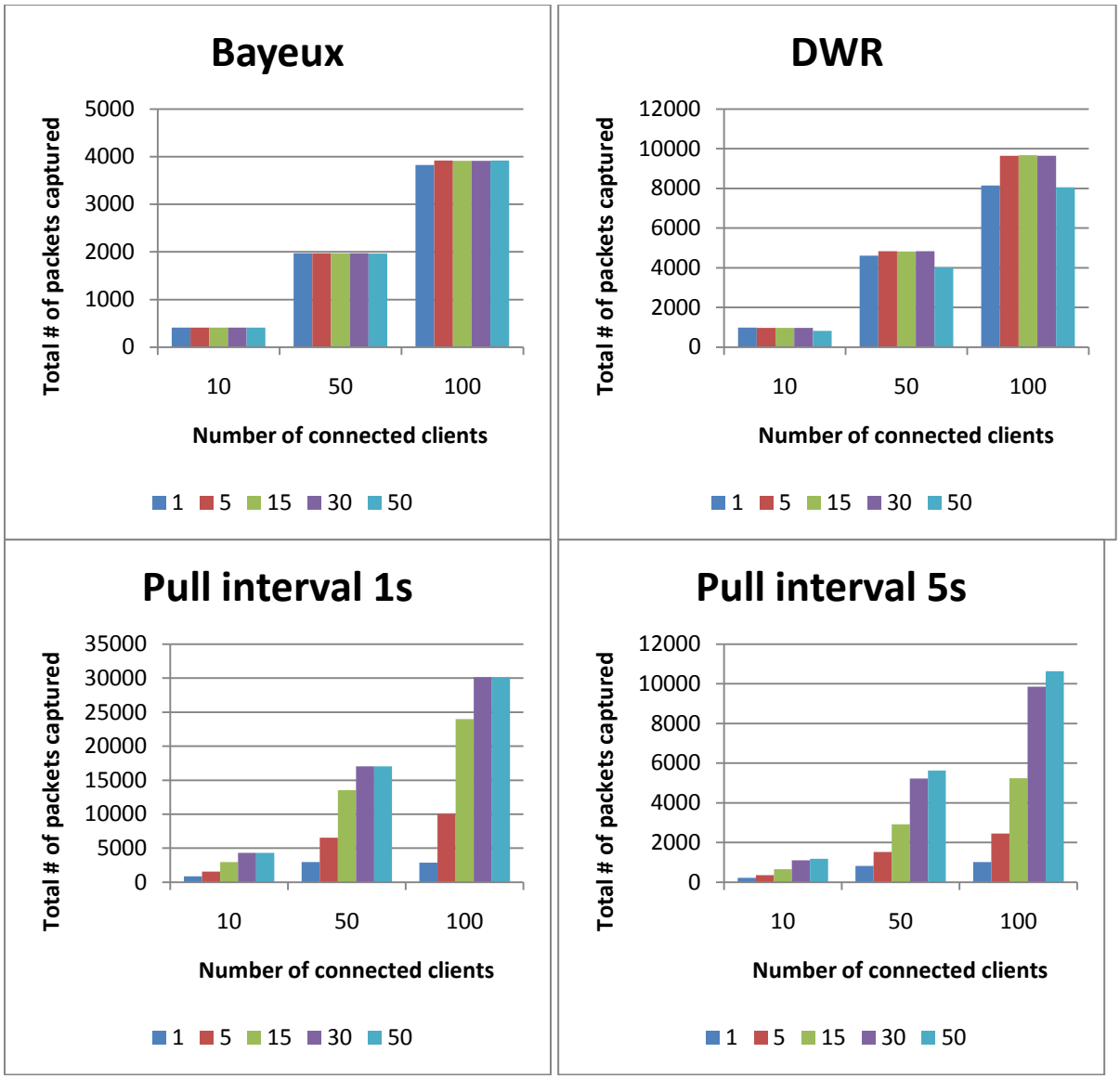


Figure 10 – Mean trip time from the moment of publishing to the moment of the client receiving the message

4.3.5 Network traffic

Figure 11 shows the amount of packets travelling between the application server and the client simulator for all tests. The best performer here is the streaming application. After the initial connections are made, very little client-side packets are sent. The amount of packets sent is roughly 14-16 times the amount of active connections. The maximum amount of packets was 1420, while there

were 100 active connections. This will be useful when network traffic is costly, for example in very congested networks or in mobile networks. The Bayeux application uses about 40 times the number of active connections with a maximum of 4098 and DWR almost doubles that with a maximum of 9667 packets. Network traffic generated by the pull application depended on the polling frequency. The higher the frequency, the more packets were sent. This is not optimal (see Sections 4.2.2). With a pull interval of 1 second, up 30132 packets were used. This dropped to 74 as the polling frequency was decreased to 50 seconds.



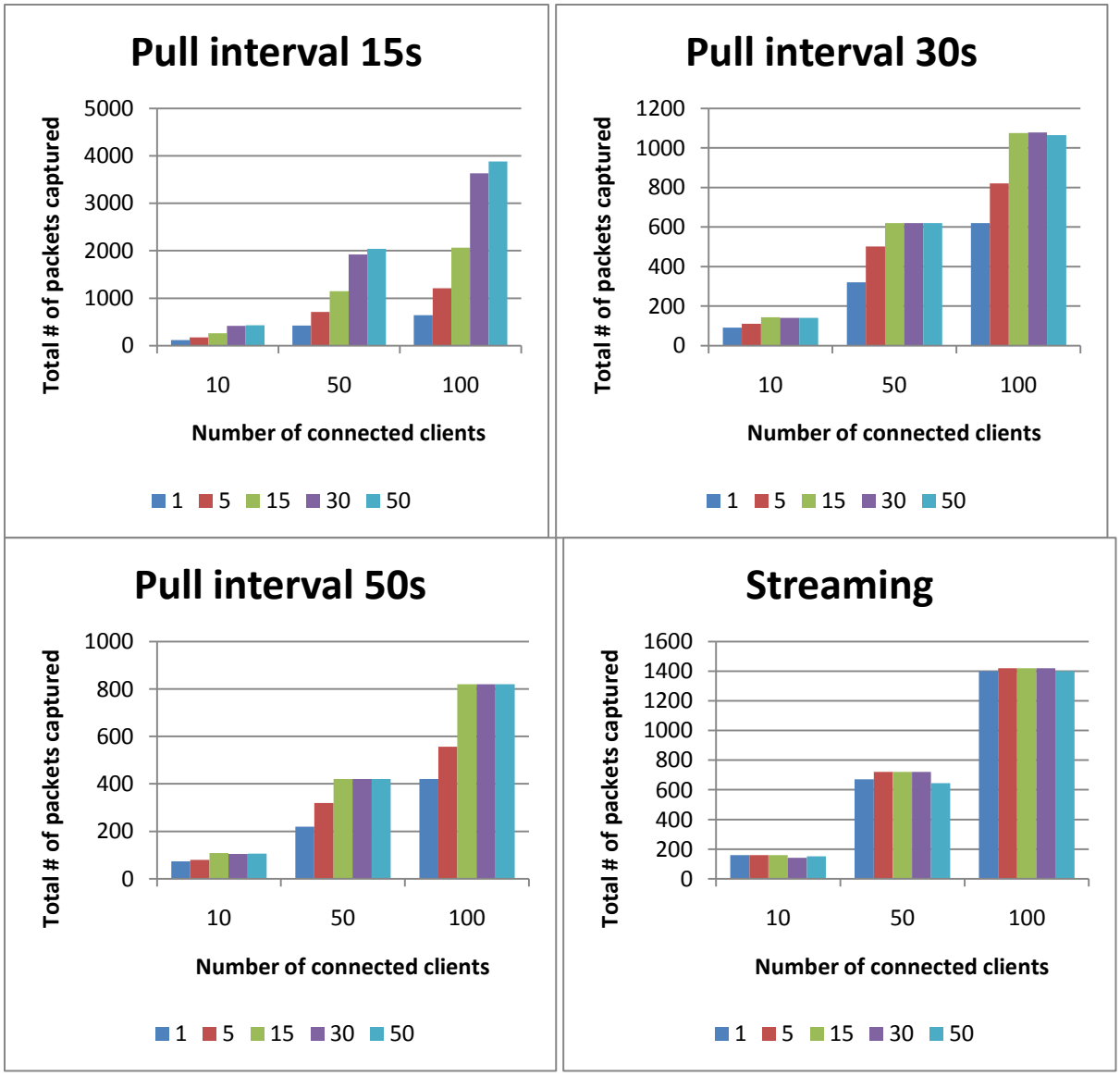


Figure 11 – total number of packages moved between the client and the server applications

4.4 Threats to Validity

4.4.1 Measurement of latency

A different method was used to measure latency in the streaming application, due to the fact that it uses a single connection for the entire test. Instead of calculating the average round trip of each message, only the last message of the series was used. This means that the data set for latency in the streaming application was a tenth of the size of the data sets for other tests.

4.4.2 Frameworks

Both long-polling applications used a framework on the server side. Bayeux used the COMETD library and DWR used its own COMET servlet. The streaming application did not use any frameworks. It is unclear how much performance optimizations within the aforementioned frameworks affect the results of these tests.

4.4.3 Continuations

It was determined during testing that Continuations as implemented in Jetty 6 were the best way of pausing a request from the client while waiting for new information. It is unclear how Jetty has implemented these Continuations and whether they suffer any performance degradation when compared to the custom pausing mechanisms, such as BayeuxContinuations, used in the long-polling solutions.

5 Conclusions

The primary goals of this thesis were to find out whether a viable way of creating HTTP streaming solutions exists and to test streaming to other ways of continuous data transfer.

During research it was found that four different ways of creating a streaming application exist: XHR streaming, WebSockets, Server Sent Events and forever frames. The first three were dismissed due to browser incompatibility and a working solution was created using the third, forever frames.

This working solution was tested against other methods of data transfer, namely polling and long-polling. The tests showed that streaming performed well in network traffic, latency, data redundancy and data coherence, but it did not outperform long-polling. In the opinion of the author, streaming did not perform well enough to justify using it in a live environment due to its shortcomings. Without a clear advantage in performance in favour of streaming and a lack of error-control among other flaws, it is better to use long-polling in any application that requires low latency and data coherency.

Bibliography

[1] **Bozdag, Engin, Mesbah, Ali and Arie, van Deursen. 2007.** *A Comparison of Push and Pull Techinques for AJAX.* Delft : Delft University of Technology, 2007.

[2] **Bozdag, Engin, Mesbah, Ali and Deursen, Arievan. 2008.** *Performance Testing of Data Delivery Techniques for AJAX Applications.* Delft : Delft University of Technology, 2008.

[3] *COMET, The Reverse AJAX Mechanism.* **Hema, Ramji. 2011.** 1, Greater Noida : International Journal of Engineering Sciences and Management, 2011, Vol. 1. 2231-3273.

[4] **Fette, Ian. 2011.** The WebSocket protocol draft-ietf-hybi-thewebsocketprotocol-09. 2011.

[5] **Fujishima, Yuzo. 2009.** Web Sockets Now Available In Google Chrome. *The Chrominium Blog.* [Online] December 9, 2009. [Cited: June 14, 2011.] http://blog.chromium.org/2009/12/web-sockets-now-available-in-google.html.

[6] **Garrett, Jesse James. 2005.** Ajax: A New Approach to Web Applications. *adaptivePath.* [Online] February 18, 2005. [Cited: June 10, 2011.] http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications.

[7] **Gutwin, Carl, Lippold, Michael and Graham, T.C. Nicholas. 2011.** *Real-Time Groupware in the Browser: Testing the Performance of Web-Based Networking.* New York : ACM, 2011.

[8] **Hickson, Ian. 2011.** Server-Sent Events. *w3.org.* [Online] June 17, 2011. [Cited: June 19, 2011.] http://dev.w3.org/html5/eventsource/.

[9] **Juvva, Kanaka and Rajkumar, Raj. 1999.** *A Real-Time Push-Pull Communications Model for Distributed Real-Time and Multimedia Systems.* Pittsburgh : Carnegie Mellon University, 1999.

[10] **Kesteren, Anne van. 2010.** Disabling the WebSocket protocol. *annevankesteren.nl.* [Online] December 8, 2010. [Cited: June 15, 2011.] http://annevankesteren.nl/2010/12/websocket-protocol-vulnerability.

[11] **Lawrence, Eric. 2010.** COMET Streaming in Internet Explorer. *MSDN.* [Online] April 5, 2010. [Cited: June 14, 2010.] http://blogs.msdn.com/b/ieinternals/archive/2010/04/06/comet-streaming-in-internet-explorer-with-xmlhttprequest-and-xdomainrequest.aspx.

[12] **Lin, Zhijie, et al. 2008.** *Research on Web Applications Using Ajax New Technologies.* Three Gorges : MultiMedia and Information Technology, 2008. 978-0-7695-3556-2.

[13] **Loreto, Salvatore, et al. 2011.** *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP.* Rome : University of Rome "Tor Vergata", 2011.

[14] **Mah, Bruce A. 1997.** *An Empirical Model of HTTP Network Traffic.* Kobe : University of California at Berkley, 1997.

[15] **Russell, Alex. 2006.** Comet: Low Latency Data for the Browser. *infrequently.* [Online] March 3, 2006. [Cited: June 10, 2011.] http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/.

[16] **Smullen, Clinton and Smullen, Stephanie. 2008.** *An Eperimental Study of AJAX Application Performance.* Chattanooga : Journal of Software, 2008.

[17] WebSockets. *MDN.* [Online] [Cited: June 15, 2011.] https://developer.mozilla.org/en/WebSockets.

[18] **2011.** When can I use Server-sent DOM events. *Caniuse.* [Online] June 17, 2011. [Cited: June 19, 2011.] http://caniuse.com/#feat=eventsource.

[19] **Wilkins, Greg. 2009.** Jetty Continuations. *codehaus.* [Online] June 9, 2009. [Cited: June 10, 2011.] http://docs.codehaus.org/display/JETTY/Continuations.

[20] **—. 2009.** Jetty WebSocket Server. *gregw.* [Online] November 24, 2009. [Cited: June 14, 2011.] http://blogs.webtide.com/gregw/entry/jetty_websocket_server.

[21] **—. 2007.** Latency: Long Polling vs Forever Frame. *Cometdaily.* [Online] December 18, 2007. [Cited: June 10, 2011.] http://cometdaily.com/2007/12/18/latency-long-polling-vs-forever-frame/.

Appendix

CHIRON

A modified version of CHIRON containing the newly created test is located on a CD that is included with this thesis. An unmodified version of CHIRON can be downloaded from http://spci.st.ewi.tudelft.nl/chiron/download.html. Version 1.2 was used in this thesis.

Instructions for running CHIRON can be found in the docs library in the file called README.txt.

Lükkamis- ja tõmbamispõhine andmete edastus AJAX rakendustes.

Bakalaureusetöö (6EAP)

Mihkel Jõhvik

Resümee

Lähiaastatel on toimunud suur liikumine veebi interaktiivsemaks muutumise suunas. Teenused nagu Gmail ja Facebook Chat on kogunud miljoneid kasutajaid ning muutunud uute veebilahenduste loomisel etalonideks.

Need rakendused teeb võimalikuks AJAX. AJAX on üldine termin erinevate tehnoloogiate kirjeldamiseks, mis võimaldavad kasutajal teha asünkroonseid päringuid serverisse ning vastuseid kasutajale kuvada. Kui aga on vaja edastada infot, mis ajas pidevalt muutub, siis AJAX rakendustel ei ole võimalik teada saada millal ja mis täpselt muutus, ilma seda pidevalt küsimata. See aga tekitab koormust nii võrgus kui ka kasutaja- ja servermasinates. Et pidevalt uut infot pärima ei peaks, on mõeldud välja COMET ehk Reverse-AJAX. Tegemist on samuti üldise terminiga, mis kirjeldab tehnoloogiaid, mis võimaldavad serveril ise kasutajatele teada anda kui info serveris muutub.

AJAX puhul on võimalikke lahendusi praktiliselt üks. Rakendus küsib iga teatud aja tagant serverist uut infot. Muutujaks on vaid pärimise sagedus.

COMET puhul on võimalikke lahendusi mitu. Kõik need nõuavad pikaajalist ühendust serveriga, kuid erinevus tekib sellest, kas sulgeda ühendus kui uus info on serverisse saabunud ja alustada kohe uut või siis kasutada ainult ühte ühendust. Esimest varianti nimetatakse *long-pollinguks* ehk pikaajaliseks päringuks. Teist *pushiks* ehk lükkamiseks.

Kasutajate jaoks on asünkroonsete rakenduste juures tähtsal kohal kiirus ning usaldusväärsus. Teenusepakkujad huvituvad aga sellest, kuidas rakendus mõjub nende infrastruktuurile.

Käesoleva bakalaureusetöö eesmärgiks oli võrrelda erinevate AJAX ja COMET realisatsioonide jõudlust. Töö baseerub Engin Bozdagi, Ali Mesbahi ja Arieva Deurseni 2008 aasta uurimustööl „Performance Testing of Data Delivery Techniques for AJAX Applications. “

Ali Mesbah et al. lõid raamistiku erinevate AJAX realisatsioonide testimiseks ning lisasid raamistikku ka vastavad tõmbamis- ja lükkamispõhised testid. Testiks oli erinevate pärimissagedustega AJAX rakendus ning kaks erinevat pikaajalise päringu rakendust.

Käesolevas töös jooksutati need testid uuesti ning lisati raamistikku uus test. Lisatud test on mõeldud lükkamispõhiste rakenduste jõudluse mõõtmiseks.

Testide tulemusena leiti, et lükkamispõhine rakendus pakub pikaajalise päringu kõrval väga head andmeedastuskiirust ning koormab vähe võrguressursse. Samas ei ole andmeedastuskiirus pikajalise päringu rakendustest märgatavalt kiirem, vaid kohati isegi aeglasem. Samuti koormas lükkamispõhine rakendus tunduvalt rohkem serverit ning ei toimetanud kogu infot kasutajani. Seega on autori arvamus, et lükkamispõhiste veebirakenduste kasutamine ei ole õigustatud.