Journal of Web Engineering, Vol. 8, No. 4 (2009) 287–315

PERFORMANCE TESTING OF DATA DELIVERY TECHNIQUES FOR AJAX APPLICATIONS

ENGIN BOZDAG ALI MESHAH
Software Engineering Research Group, Delft University of Technology
Mekelweg 4, 2628CD Delft, The Netherlands
bozdag@gmail.com A.Mesbah@tudelft.nl

ARIE van DEURSEN
Software Engineering Research Group, Delft University of Technology & CWI
Mekelweg 4, 2628CD Delft, The Netherlands
Arie.vanDeursen@tudelft.nl

Received February 15, 2008
Revised September 6, 2009

AJAX applications are designed to have high user interactivity and low user-perceived latency. Real-time dynamic web data such as news headlines, stock tickers, and auction updates need to be propagated to the users as soon as possible. However, AJAX still suffers from the limitations of the Web's request/response architecture which prevents servers from pushing real-time dynamic web data. Such applications usually use a pull style to obtain the latest updates, where the client actively requests the changes based on a predefined interval. It is possible to overcome this limitation by adopting a push style of interaction where the server broadcasts data when a change occurs on the server side. Both these options have their own trade-offs. This paper first introduces the characteristics of both pull and push approaches. It then presents the design and implementation of our distributed test framework, called CHIRON, where different AJAX applications based on each approach can be automatically tested on. Finally, we present and discuss the results of our empirical study comparing different web-based data delivery approaches.

Keywords: Performance testing, Ajax, Web data delivery, Comet, Push/pull, Empirical study

1 Introduction

Recently, there has been a shift in the direction of web development. A new breed of web application, dubbed AJAX (Asynchronous JavaScript and XML) [14] is emerging in response to the limited degree of interactivity in large-grain stateless Web interactions. The intent is to make web pages feel more responsive by exchanging small amounts of data with the server behind the scenes and making changes to individual user interface components. This way, the entire web page does not have to be reloaded each time the user makes a change. AJAX is a serious option not only for newly developed applications, but also for existing web sites if their user friendliness is inadequate [21, 22, 20, 23].

The term AJAX spread rapidly from a Weblog to the Wall Street Journal within weeks. The new web applications under the AJAX banner have redefined end users' expectations of what is possible within a Web browser. However, AJAX still suffers from the limitations of the Web's request/response architecture. The classical model of the web called REST [12] requires all communication between the browser and the server to be initiated by the client, i.e., the end user clicks on a button or link

and thereby requests a new page from the server. No 'permanent' connection is established between client/server and the server is required to maintain no state information about the clients. This "pull" scheme helps scalability [12], but precludes servers from sending asynchronous notifications. There are many use cases where it is important to update the client user interface in response to server-side changes. For example:

- An auction web site, where the users need to be averted that another bidder has made a higher bid. Figure 1 shows a screen-shot taken from eBay. In a site such as eBay, the user has to continuously press the 'refresh' button of his or her browser, to see if somebody has made a higher bid.
- A stock ticker, where stock prices are frequently updated. Figure 2 shows a screen-shot taken from MSN's MoneyCentral site[a] . The right column contains a stock ticker. The site currently uses a *pull*-based mechanism to update the stock data.
- A chat application, where new sent messages are delivered to all the subscribers.
- A news portal, where news items are pushed to the subscriber's browser when they are published.

Today, such web applications requiring real-time *event notification* and *data delivery* are usually implemented using a *pull* style, where the client component actively requests the state changes using client-side timeouts. An alternative to this approach is the *push*-based style, where the clients subscribe to their topic of interest, and the server publishes the changes to the clients asynchronously every time its state changes.

However, implementing such a push solution for web applications is not trivial, mainly due to the limitations of the HTTP protocol. It is generally accepted that a push solution that keeps an open connection for all clients will cause scalability problems. However, as far as we know, no empirical study has been conducted into the actual trade-offs involved in applying a push- versus pull-based approach to browser-based or AJAX applications. Such a study will answer questions about data coherence, scalability, network usage and latency. It will also allow web engineers to make rational decisions concerning key parameters such as publish and pull intervals, in relation to, e.g., the anticipated number of clients.

In this paper, which is an extension of our previous work [5], we focus on the following challenges:

- How can we set up an automated, controllable, repeatable, and distributed test environment, so that we can obtain empirical data with high accuracy for comparing data delivery approaches for AJAX applications?
- How does a push-based web data delivery approach compare to a pull-based one, in terms of data coherence, scalability, network performance, and reliability?

This paper is further organized as follows.

We start out, in Section 2, by exploring current techniques for real-time HTTP-based data delivery on the web. Subsequently, in Section 3, we discuss the push-based BAYEUX protocol and the DWR library, the two open source push implementations that we will use in our experiments. In Section 4,

[a] `http://moneycentral.msn.com`

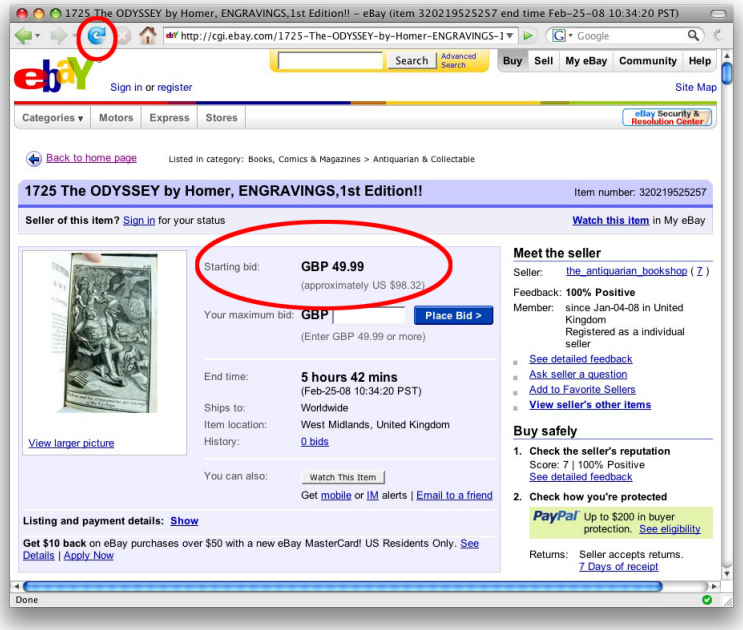

Fig. 1. A screenshot taken from eBay. The user has to constantly click the "Refresh" button to see any updates.

we present the experimental design by articulating our research questions and outlining the proposed approach. The independent and dependent variables of our experiment are also discussed in this section. A detailed presentation of our distributed testing framework called CHIRON[b] as well as the environment and applications that we use to conduct our experiments, is shown in Section 5. In Section 6 the results of our empirical study involving push and pull data delivery techniques are covered, followed by a discussion of the findings of the study and threats to validity in Section 7. Finally, in Section 8, we survey related work on this area, after which we conclude our paper in Section 9 with a summary of our key contributions and suggestions for future work.

2 Web-based Real-time Event Notification

2.1 AJAX

AJAX [14] is an approach to web application development utilizing a combination of established web technologies: standards-based presentation using XHTML and CSS, dynamic display and interaction using the Document Object Model, data interchange and manipulation, asynchronous data retrieval using XMLHttpRequest, and JavaScript binding everything together. XMLHttpRequest is an API implemented by most modern web browser scripting engines to transfer data to and from a web server using HTTP, by establishing an independent communication channel in the background between a web client and server.

It is the combination of these technologies that enables us to adopt principal software engineering paradigms, such as component- and event-based approaches, for web application development. Our earlier work [21] on an architectural style for AJAX, called SPIAR, gives an overview of the new way

[b] In Greek mythology, CHIRON, was the only immortal centaur. He became the tutor for a number of heroes, including AJAX.

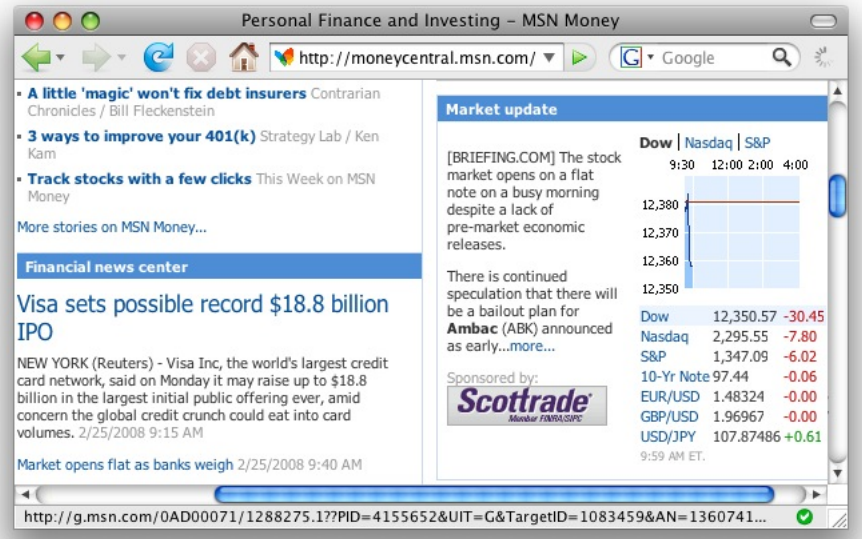

Fig. 2. Stock ticker from MSN MoneyCentral

web applications can be architected using AJAX. The evolution of web and the advent of *Web 2.0*, and AJAX in particular, is making the users experience similar to using a desktop application. Well known examples include Gmail[c] , Google Maps[d] , or Google Documents[e] , in which, for example, multiple users can edit the same spreadsheet simultaneously, all via their web browser.

The classical page-sequence web, based on the REST style, makes a server-initiated HTTP request impossible. Every request has to be initiated by a client, precluding servers from sending asynchronous notifications without a request from the client [18]. There are several solutions used in practice that still allow the client to receive (near) real-time updates from the server. In this section we analyze some of these solutions.

2.2 HTTP Pull

Most web applications check with the server at regular user-definable intervals known as *Time to Refresh* (TTR). This check occurs blindly regardless of whether the state of the application has changed.

In order to achieve high data accuracy and data freshness, the pulling frequency has to be high. This, in turn, induces high network traffic and possibly unnecessary messages. The application also wastes some time querying for the completion of the event, thereby directly impacting the responsiveness to the user. Ideally, the pulling interval should be equal to the *Publish Rate* (PR), i.e., rate at which the state changes. If the frequency is too low, the client can miss some updates.

This scheme is frequently used in web systems, since it is robust, simple to implement, allows for offline operation, and scales well to high number of subscribers [15]. Mechanisms such as Adaptive TTR [4] allow the server to change the TTR, so that the client can pull on different frequencies,

[c] http://www.gmail.com
[d] http://maps.google.com
[e] http://docs.google.com

depending on the change rate of the data. This dynamic TTR approach in turn provides better results than a static TTR model [28]. However, it will never reach complete data accuracy, and it will create unnecessary traffic.

2.3 HTTP Streaming

HTTP Streaming is a basic and old method that was introduced on the web first in 1992 by Netscape, under the name 'dynamic document' [24]. HTTP Streaming comes in two forms namely, Page and Service Streaming.

2.3.1 Page Streaming

This method simply consists of streaming server data in the response of a long-lived HTTP connection. Most web servers do some processing, send back a response, and immediately exit. But in this pattern, the connection is kept open by running a long loop. The server script uses event registration or some other technique to detect any state changes. As soon as a state change occurs, it streams the new data and flushes it, but does not actually close the connection. Meanwhile, the browser must ensure the user-interface reflects the new data, while still waiting for response from the server to finish. Implementation of this

2.3.2 Service Streaming

Service Streaming relies on the XMLHttpRequest object. This time, it is an XMLHttpRequest connection that is long-lived in the background, instead of the initial page load. This brings some flexibility regarding the length and frequency of connections. The page will be loaded normally (once), and streaming can be performed with a predefined lifetime for connection. The server will loop indefinitely just like in page streaming, and the browser has to read the latest response (*responseText*) to update its state on the Document Object Model (DOM).

2.4 Comet or Reverse AJAX

Currently, major AJAX push tools support Service Streaming. The application of the Service Streaming scheme under AJAX is now known as Reverse AJAX or COMET [25]. COMET enables the server to send a message to the client when the event occurs, without the client having to explicitly request it. Such a client can continue with other work while expecting new data from the server. The goal is to achieve a real-time update of the state changes and offer a solution to the problems mentioned in Section 2.2.

The COMET scheme is available thanks to the 'persistent connection' feature brought by HTTP/1.1. With HTTP/1.1, unless specified otherwise, the TCP connection between the server and the browser is kept alive, until an explicit 'close connection' message is sent by one of the parties, or a timeout/network error occurs. Prior to persistent connections, a separate TCP connection was established to fetch each URL, increasing the load on HTTP servers and causing congestion on the Internet. With persistent connections, fewer TCP connections are opened and closed, leading to savings both in CPU time for routers and hosts (clients, servers, proxies, gateways, tunnels, or caches), as well as in memory usage for TCP protocol control blocks in hosts.

HTTP/1.1 reduces the total number of TCP connections in use. However, it still states that the protocol must follow the request/response scheme, where the client makes a request and the server returns a response for this particular request. Thus, once a complete response is returned, there is no further way for the server to send data back to the client browser.

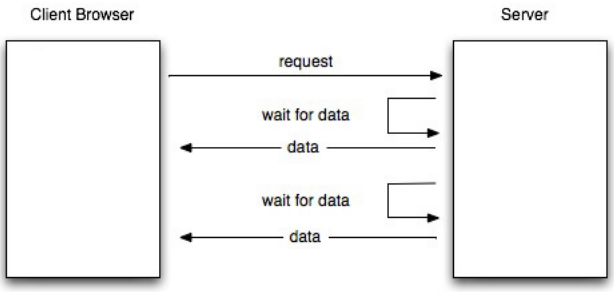

Fig. 3. Streaming mode for COMET

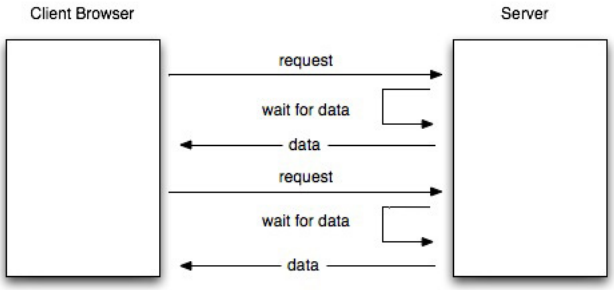

Fig. 4. Long polling mode for COMET

COMET implementations have mainly adopted the following techniques to overcome this problem:

Streaming Figure 3 shows how the streaming mode operates. After the initial request, the server does not close the connection, nor does it give a full response. As the new data becomes available, the server returns it to the client in HTTP chunked mode [32], using the same request and the connection. Typical Comet server implementations implement this feature by opening a hidden *iframe* element in the browser after page load, establishing a long-lived connection inside the hidden iframe. Data is pushed incrementally from the server to the client over this connection, and rendered incrementally by the web browser [27].

The problem with this approach is that some web servers might identify this open-request as idle and close the connection. HTTP chunked mode might also be not supported in every router that is located on the path.

Long polling Figure 4 shows the operation of the long polling mode. In this mode, once again the server holds on to the client request, however this time until data becomes available. If an event occurs, the server sends the data to the client and the client has to reconnect. Otherwise, the server holds on to the connection for a finite period of time, after which it asks the client to reconnect again. Long polling (also known as Asynchronous-Polling) is a mixture of pure server push and client pull. If the publish interval (or the timeout value) is low, the system acts more like a pure pull-based style. If the publish interval is high, it will act more like a pure push approach.

3 COMET Implementations

COMETD and DWR are currently two actively developed open source libraries that bring COMET support to AJAX applications. In the following subsections we take a closer look at these two libraries.

3.1 COMETD *Framework and the* BAYEUX *Protocol*

As a response to the lack of communication standards [21] for AJAX applications, the COMETD group[f] released a COMET protocol draft called BAYEUX [26]. The BAYEUX message format is defined in JSON (JavaScript Object Notation)[g] which is a data-interchange format based on a subset of the JavaScript Programming Language. The protocol has recently been implemented and included in a number of web servers including Jetty[h] and IBM Websphere[i] .

This protocol follows the 'topic-based' [11] publish-subscribe scheme, which groups events according to their topic (name) and maps individual topics to distinct communication channels. Participants subscribe to individual topics, which are identified by keywords. Like many modern topic-based engines, BAYEUX offers a form of *hierarchical addressing*, which permits programmers to organize topics according to containment relationships. It also allows topic names to contain *wildcards*, which offers the possibility to subscribe and publish to several topics whose names match a given set of keywords. BAYEUX defines the following phases in order to establish a COMET connection:

1. The client performs a handshake with the server, receives a client ID and list of supported connection types, such as IFrame or long-polling (See Section 2.4).
2. The client sends a connection request with its ID and its preferred connection type.
3. The client later subscribes to a channel and receives updates

Although the BAYEUX specification supports both streaming and long polling modes, the COMETD framework currently only implements the long polling approach. Please note that, although there are other frameworks that support the BAYEUX protocol, in this paper we will use the terms BAYEUX and COMETD interchangeably.

3.2 *Direct Web Remoting (*DWR*)*

Direct Web Remoting (DWR)[j] is a Java open source library which allows scripting code in a browser to use Java functions running on a web server just as if they were in the browser. DWR works by dynamically generating JavaScript based on Java classes. To the user it feels like the execution is taking place on the browser, but in reality the server is executing the code and DWR is marshalling the data back and forwards. DWR works similar to the RPC mechanism (e.g., Java RMI [29]), but without requiring any plugins. It consists of two main parts:

- A Java Servlet running on the server that processes requests and sends responses back to the browser.
- A JavaScript engine running in the browser that sends requests and can dynamically update the DOM with received responses from the server.

[f] http://www.cometd.com
[g] http://www.json.org
[h] http://www.mortbay.org
[i] http://www.ibm.com/software/websphere
[j] http://getahead.org/dwr

From version 2.0 and above DWR supports COMET and calls this type of communication "Active Reverse AJAX" [10]. Currently, DWR does not support BAYEUX, and has adopted its own protocol to exchange data. DWR supports the *long polling* as well as the *streaming* mode. Because COMETD has no streaming implementation, we will only use the long polling mode of both libraries in our experiment, in order to be able to make a comparison.

4 Experimental Design

In this section we first present our research questions. Later we describe our proposed approach, and the independent and dependent variables which we will use to conduct the experiment and come to answers to our questions.

4.1 Goal and Research Questions

We set up our experiments in order to evaluate several dependent variables using the GQM/MEDEA[k] framework proposed by Briand *et al.*[6]. First, we describe the goal, perspective and environment of our experiment:

Goal: To obtain a rigorous understanding of the actual performance trade offs between a push-based and a pull-based approach to AJAX data delivery.

Perspective: In particular, we aim at an automated, repeatable experiment, in which as many (combinations) of the factors that influence performance (such as the number of users, the number of published messages, the intervals between messages, etc.) can be taken into account.

Environment: The experiments are targeted at distributed environments, particulary those with UNIX/Linux nodes. Further relevant factors of these systems will be described in Section 5.2.

We have formulated a number of questions that we would like to find answers for in each approach. Our research questions can be summarized as:

RQ1 How fast are state changes (new messages) on the server propagated to the clients?

RQ2 What is the scalability and overall performance of the server?

RQ3 To what extent is the network traffic generated between the server and clients influenced?

RQ4 How reliable is each approach? Are there messages that are missed or abundantly received on the clients?

4.2 Outline of the Proposed Approach

In order to obtain an answer to our research questions, we propose the following steps:

1. creating two separate web applications, having the same functionality, using each push library (one for COMETD and one for DWR), consisting of the client and the server parts,
2. creating a pull-based web application with the same functionality as the other two,
3. implementing an application, called *Service Provider*, which publishes a variable number of data items at certain intervals,

[k] Goal Question Metric/MEtric DEfinition Approach

4. simulating a variable number of concurrent web users operating on each application, by creating *virtual users*,

5. gathering data and measuring: the mean time it takes for clients to receive a new published message, the load on the server, number of messages sent or retrieved, and the effects of changing the data publish rate and number of users,

6. coordinating all these tools automatically in order to have consistent test runs for each combination of the variables,

7. reporting, analyzing, and discussing the measurements found.

4.3 Independent Variables

To see how the application server reacts to different conditions, we use different combinations of the following independent variables:

Number of concurrent users 100, 500, 1000, 2000, 5000 and 10000; the variation helps to find a maximum number of users the server can handle simultaneously.

Publish interval 1, 5, 15, 30, and 50 seconds; the frequency of the publishing updates is also important. Because of the *long polling* implementation in COMETD and DWR (See Section 2), the system should act more like pure pull when the publish interval is small, and more like pure push when the publish interval increases. This is because a smaller publish interval causes many reconnects (See Section 2.4).

Pull interval 1, 5, 15, 30, and 50 seconds; when a pull approach is used, the pulling interval will also have an effect on the measurements.

Application mode COMETD, DWR, and pull; we also made an option in our framework that allows us to switch between different application modes.

Total number of messages 10; to constrain the period needed to conduct the experiment, for each test run, we generate a total of 10 publish messages.

4.4 Dependent Variables

In order to be able to answer the research questions we measure the following dependent variables:

Mean Publish Trip-time (MPT) We define trip-time as follows:

$$\textit{Trip-time} = | \textit{Data Creation Date} - \textit{Data Receipt Date} |$$

Data Creation Date is the date on the publishing server the moment it creates a message, and *Data Receipt Date* is the date on the client the moment it receives the message. Trip-time shows how long it takes for a publish message to reach the client and can be used to find out how fast the client is notified with the latest events.For each combination of the independent variables, we calculate the mean of the publish trip-time for the total number of clients.

We define a piece of data as *coherent*, if the data on the server and the client is synchronized. We check the data coherence of each approach by measuring the trip-time. Accordingly, a data item with a low trip-time leads to a high coherence degree.

Server Performance (SP) Since push is stateful, we expect it to have more administration costs on the server side, using more resources. In order to compare this with pull, we measure the CPU usage for the push- and pull-based approaches.

Received Publish Messages (RPM) To see the message overhead, we publish a total of 10 messages and count the total number of (non unique) messages received by the clients that could make a connection to the server. This shows us if a client receives an item multiple times and causes unnecessary network traffic.

Received Unique Publish Messages (RUPM) It is also interesting to see if all the 10 messages we have published reach the clients that could make a connection to the server. This shows us if a client misses any items.

Received Message Percentage (RMP) It is quite possible that not all the clients could make a connection, or receive all the messages. Therefore, for each run, we divide the total number of messages received, by the total number of messages published.

Network Traffic (NT) In order to see the difference in network traffic, we record the number of TCP packets coming to and going from the server to the clients.

5 Distributed Testing Infrastructure

In order to measure the impact the various combinations of independent variables have on the dependent variables, we need a distributed testing infrastructure. This infrastructure must make it possible to control independent variables such as the number of concurrent users and total number of messages, and observe the dependent variables, such as the trip-time and the network traffic.

Unfortunately, distributed systems are inherently more difficult to design, program, and test than sequential systems [2]. They consist of a varying number of processes executing in parallel. A process may also update its variables independently or in response to the actions of another process. Testing distributed programs is a challenging task of great significance. *Controllability*, *observability* [8], and *reproducibility* problems might occur in distributed testing environments. In this section we will present our distributed, automated testing framework called CHIRON and how it helps to decrease the magnitude of these problems. We are in the process of releasing our testing framework under an open source license. More information about CHIRON can be obtained from the following URL: `http://spci.st.ewi.tudelft.nl/chiron/`

5.1 The CHIRON *Distributed Testing Framework*

As mentioned in Section 4.3, we defined several independent variables in order to measure several dependent variables (see Section 4.4). The combination of the independent variables (i.e. pull intervals, publish intervals, and the number of users) is huge and that makes performing the tests manually an error-prone, and time consuming task.

In addition, the tools and components we use are located in different machines to simulate real-world environments. This distributed nature contributes to the complexity of controlling the experiment manually. This all makes it difficult to repeat the experiment at later stages with high validity.

In order to overcome these challenges, we have created an integrated performance testing framework called CHIRON that automates the whole testing process. As depicted in Figure 5, the controller has direct access to different servers and components (Application server, client simulation server,

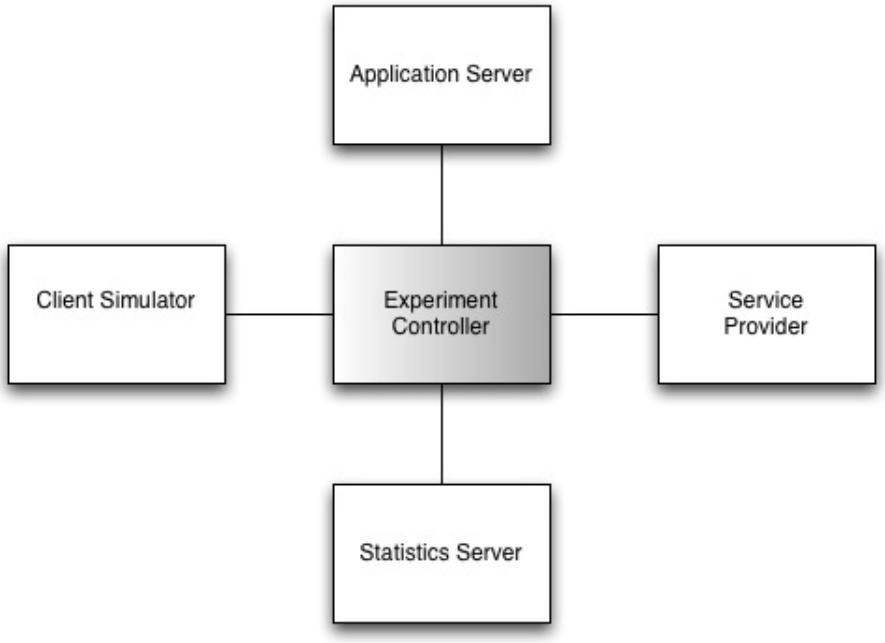

Fig. 5. Design of the Distributed Automated Testing Framework CHIRON

the statistic server and the Service Provider). By automating each test run, the controller coordinates the whole experiment. This way we can repeat the experiment many times without difficulty and reduce the non-determinism, which is inherent in distributed systems [2]. Since no user input is needed during a test run, observability and controllability problems [8] are minimized.

We have implemented CHIRON using a number of open source packages. In particular we use *Grinder*[1] which seemed to be a good option, providing an internal TCPProxy, allowing to record and replay events sent by the browser. It also provides scripting support, which allows us to create a script that simulates a browser connecting to the push server, subscribing to a particular stock channel and receiving push data continuously. In addition, Grinder has a built-in feature that allows us to create multiple threads of a simulating script.

Figure 6 shows the steps CHIRON follows for each test iteration. The middle column denotes the task CHIRON performs, the left column denotes the step number and the right column shows which machine CHIRON interacts with in order to perform the task. For each combination of the independent variables, CHIRON carries out the tasks 3–10:

1. Read the input from the configuration file. This input consists of the independent variables, but also local and remote folders on different servers, path to the tools, publish channels, number of nodes that are used to simulate clients, etc,

2. Start the statistics server to listen for and receive experimental data from the clients,

[1] http://grinder.sourceforge.net

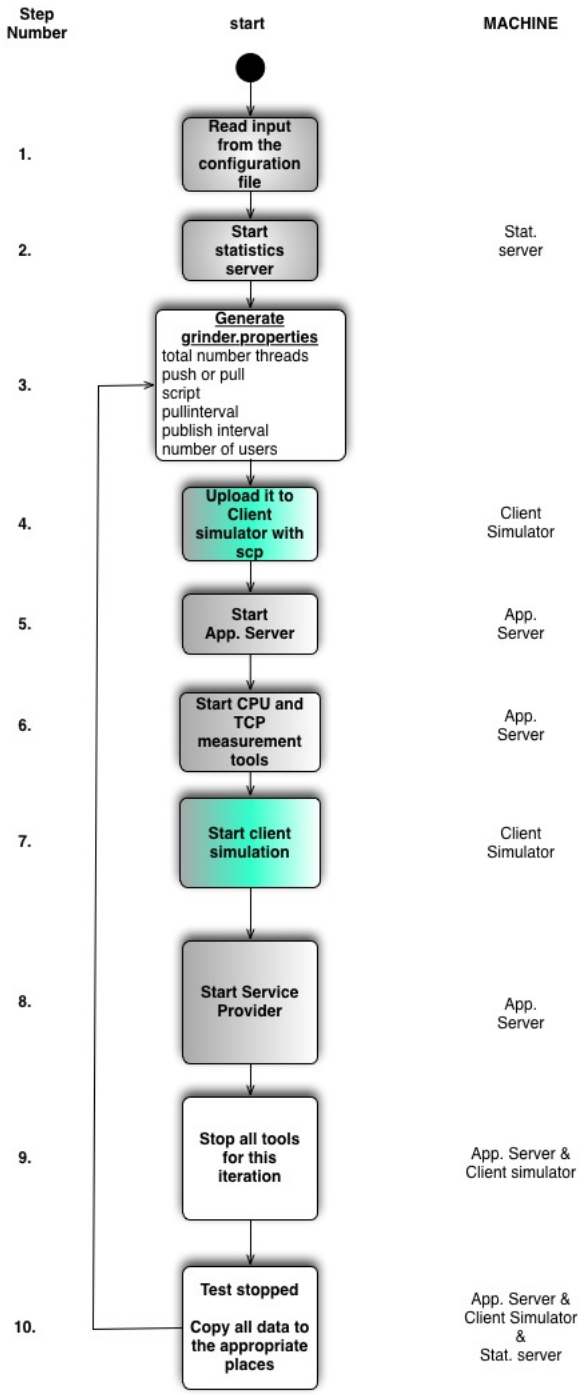

Fig. 6. The steps CHIRON follows for each iteration

3. Generate a Grinder properties file for this iteration with a combination of independent variables (pull interval, publish interval, number of users, etc.),
4. Upload the created properties file to the client simulation server,
5. Start the application server that hosts the three different versions (See Section 5.4) of the Stock Ticker web application,
6. Start performance and network traffic measurement tools on the application server,
7. Start simulating clients using the generated and uploaded properties file,
8. Start the publisher and begin publishing data to the server,
9. When the publisher is done or a time-out has occurred, stop all the components,
10. Gather and copy the data to the statistics server and get ready for the next iteration by going to the Grinder properties generation step.

Because of the distributed nature of the simulated clients, we use Log4J's SocketServer[m] to set up a logging server that listens for incoming log messages. The clients then send the log messages using the `SocketAppender`.

We use *TCPDump*[n] to record the number of TCP (HTTP) packets sent to and from the server on a specific port. Note that we only record packets coming/going to the simulated clients. We also have created a script that uses the UNIX *top*[o] utility to record the CPU usage of the application server every second. This is necessary to observe the scalability and performance of each approach.

Finally, we use Trilead's SSH2 library to manage (start, stop, etc) all the tools mentioned above. Trilead SSH-2 for Java[p] is an open source library which implements the SSH-2 protocol in Java. Since the tools are distributed on different machines, SSH2 library allows us to automatically issue commands from one single machine, gather the results on different machines and insert them into our statistics server.

In addition, we have created a *Data Analyzer* component that parses log files of the different tools and serializes all the data into a database using Hibernate[q] and MySQL Connector/J[r] . This way, different views on the data can be obtained easily using queries to the database.

5.2 Testing Environment

We use the Distributed ASCI Supercomputer 3 (DAS3)[s] to simulate the virtual users on different distributed nodes. The DAS3 cluster at Delft University of Technology consists of 68 dual-CPU 2.4 GHz AMD Opteron DP 250 compute nodes, each having 4 GB of memory. The cluster is equipped with 1 and 10 Gigabit/s Ethernet, and runs Scientific Linux 4. It is worth noting that we use a combination of the 64 DAS3 nodes and Grinder threads to simulate different numbers of users.

[m] http://logging.apache.org/log4j/docs/
[n] http://www.tcpdump.org/
[o] http://www.unixtop.org/
[p] http://www.trilead.com/Products/Trilead-SSH-2-Java/
[q] http://www.hibernate.org
[r] http://www.mysql.com/products/connector/j/
[s] http://www.cs.vu.nl/das3/overview.shtml

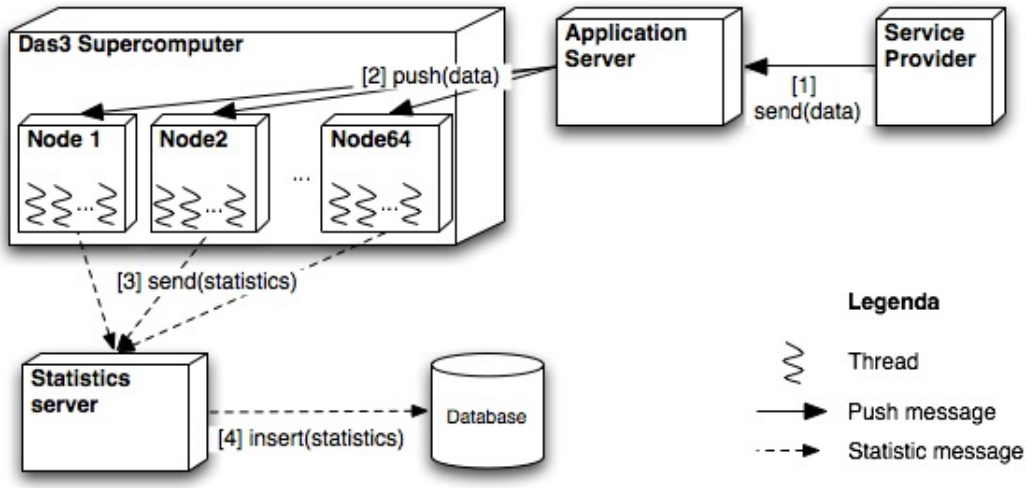

Fig. 7. Sequence of Events in the Experimental Environment.

The application server runs on a AMD 2x Dual Core Opteron 2212 machine with 8 GB RAM. The server has Ubuntu 7.10 server edition installed. For the *push* version we use the COMETD *6.1.7* and DWR *2.0.2* libraries. Both libraries run on Jetty[t] , an open source web server implemented entirely in Java. Jetty uses Java's new IO package (NIO). The NIO package follows the event-driven design, which allows the processing of each task as a finite state machine (FSM). As the number of tasks reach a certain limit, the excess tasks are absorbed in the server's event queue. The throughput remains constant and the latency shows a linear increase. The Event-driven design is supposed to perform significantly better than thread-concurrency model [33, 34]. Jetty also contains a mechanism called Continuations [9]. This mechanism allows an HTTP request to be suspended and restarted after a timeout or after an asynchronous event has occurred. This way less threads are occupied on the server. Note that no other process was running in this application server other than TCPDump and UNIX top. Their load on the server and their effect on the results are negligible, since we only measure the PID of Jetty. Besides, as mentioned, the test machine has 2x Dual Core processors, and the server never had 100% load, which can be seen in Section 6.2.

The connectivity between the server and DAS3 nodes is through a 100 Mbps ethernet connection.

5.3 Example Scenario

Figure 7 shows the sequence of events of an example test scenario from the perspective of the servers:

1. The Service Provider publishes the stock data to the application server via an HTTP POST request, in which the creation date, the stock ID, and the stock message are specified.

2. For push: The application server *pushes* the data to all the subscribers of that particular stock. For pull: the application server updates the internal stock object, so that when clients send pull requests, they get the latest data.

[t] www.mortbay.org

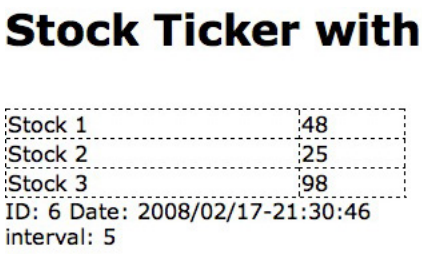

Fig. 8. Sample Stock Ticker Application

3. Each simulated client logs the responses (after some calculation) and sends it to the statistics server.

5.4 Sample Application: Stock Ticker

We have developed a Stock Ticker web application as depicted in Figure 8. As new stock messages come in, the fields are updated in the browser accordingly. As mentioned before, the Stock Ticker has been implemented in three variants, namely, COMETD, DWR, and pull.

The COMETD version consists of a JSP page which uses Dojo's COMETD library[u] to subscribe to a channel on the server and receive the Stock data. For the server side, we developed a Java Servlet (`PushServlet`) that pushes the new data into the clients' browsers using the COMETD library.

The DWR version consists of the same JSP page as the COMETD version, but uses the DWR library instead of Dojo on the client side. For the server side, we again have a separate Java Servlet (PushServlet) that pushes the data into the browsers, but this time it uses the DWR's COMET servlet.

The pull version also consists of a JSP page, but instead of COMETD or DWR, it uses the normal *bind* method of Dojo to request data from the server. The pull nature is set using the standard `setInterval` JavaScript method. The interval at which the client should request/pull for new updates is configurable. On the server, a `PullServlet` is created which updates and keeps an internal stock object (the most recent one) and simply handles and responds to every incoming request the classical way.

The Service Provider uses the HTTPClient library[v] to publish stock data to the `Servlets`. The number of publish messages as well as the interval at which the messages are published are configurable.

Concurrent clients are simulated by using the Grinder `TCPProxy` to record the actions of the JSP client pages for push and pull and create scripts for each in Jython[w]. Jython is an implementation of the high-level, dynamic, object-oriented language Python, integrated with the Java platform. It allows the usage of Java objects in a Python script and is used by Grinder to simulate web users. In our framework, we created Jython scripts that are actually imitating the different versions of the client pages.

[u] `http://dojotoolkit.org/`
[v] `http://jakarta.apache.org/commons/httpclient/`
[w] `http://www.jython.org`

6 Results and Evaluation

In the following subsections, we present and discuss the results which we obtained using the combination of variables mentioned in Section 4.3. Figures 9–14 depict the results. For each number of clients on the *x-axis*, the five publish intervals in seconds (1, 5, 15, 30, 50) are presented as colored bars. Note that the scale on the *y-axis* might not be the same for all the graphics.

6.1 Mean Publish Trip-time and Data Coherence

Figure 9 shows the mean publish trip-time versus the total number of clients for each publish interval, with COMETD (shown with BAYEUX label), DWR and *pull* techniques.

As we can see in Figure 9, Mean Publish Trip-time (MPT) is, at most, 1200 milliseconds with COMETD, which is significantly better than the DWR and *pull* approaches. Surprisingly, DWR's performance is worse than *pull* with a pull interval of 1. However, with a pull interval of 5 or higher, DWR performs better (with the exception of: clients = 2000, publish interval = 30) than *pull*. Also note that MPT is only calculated with the trip-time data of the clients that actually received the items. As we will see in Section 6.4, pull clients may miss many items, depending on the pull interval. From these results we can say that pull has a lower degree of data coherence compared to COMETD, even with a very small pull interval.

6.2 Server Performance

Figure 10 shows the mean percentage of the server CPU usage. In all scenarios, we see that the server load increases as the number of users increases. With pull, even with a pull interval of 1, the CPU usage is lower than COMETD and DWR.

However, with push, we notice that even with 10000 users, the server is not saturated: CPU usage reaches 45% with COMETD, and 55% with DWR. We assume that this is partly due to the Jetty's continuation mechanism (See Section 5.2).

6.3 Received Publish Messages

Figure 11 shows the mean number of received non-unique publish items versus the total number of clients. Note that this shows the mean of Received Publish Messages (RPM) for only those clients that could make a connection to the server and receive data. Also note that if a pull client makes a request while there is no new data, it will receive the same item again and this pattern can happen multiple times. This way a client might receive more than 10 messages. As we mentioned in Section 2.2, in a pure pull system, the pulling frequency has to be high to achieve high data coherence. If the frequency is higher than the data publish interval, the pulling client will pull the same data more than once, leading to redundant data and overhead.

We notice that with a pull interval of 1, the clients can receive up to approximately 250 non-unique messages, while we published only 10. In the same figure we see that push clients (both COMETD and DWR) received approximately a maximum of 10 messages. This means that, in the worst-case (`pull interval = 1, publish interval = 50`) 96% of the total number of pull requests were unnecessary. In the COMETD and DWR graphics, we see that, with up to 2000 users the server is very stable; almost all the clients receive a maximum of 10 published messages. What is interesting, however, as the number of clients becomes 2000 or more, some data miss begins to occur for both push approaches. Pull clients also begin to miss some data with 2000 users or more, but the decrease in received data items is much less.

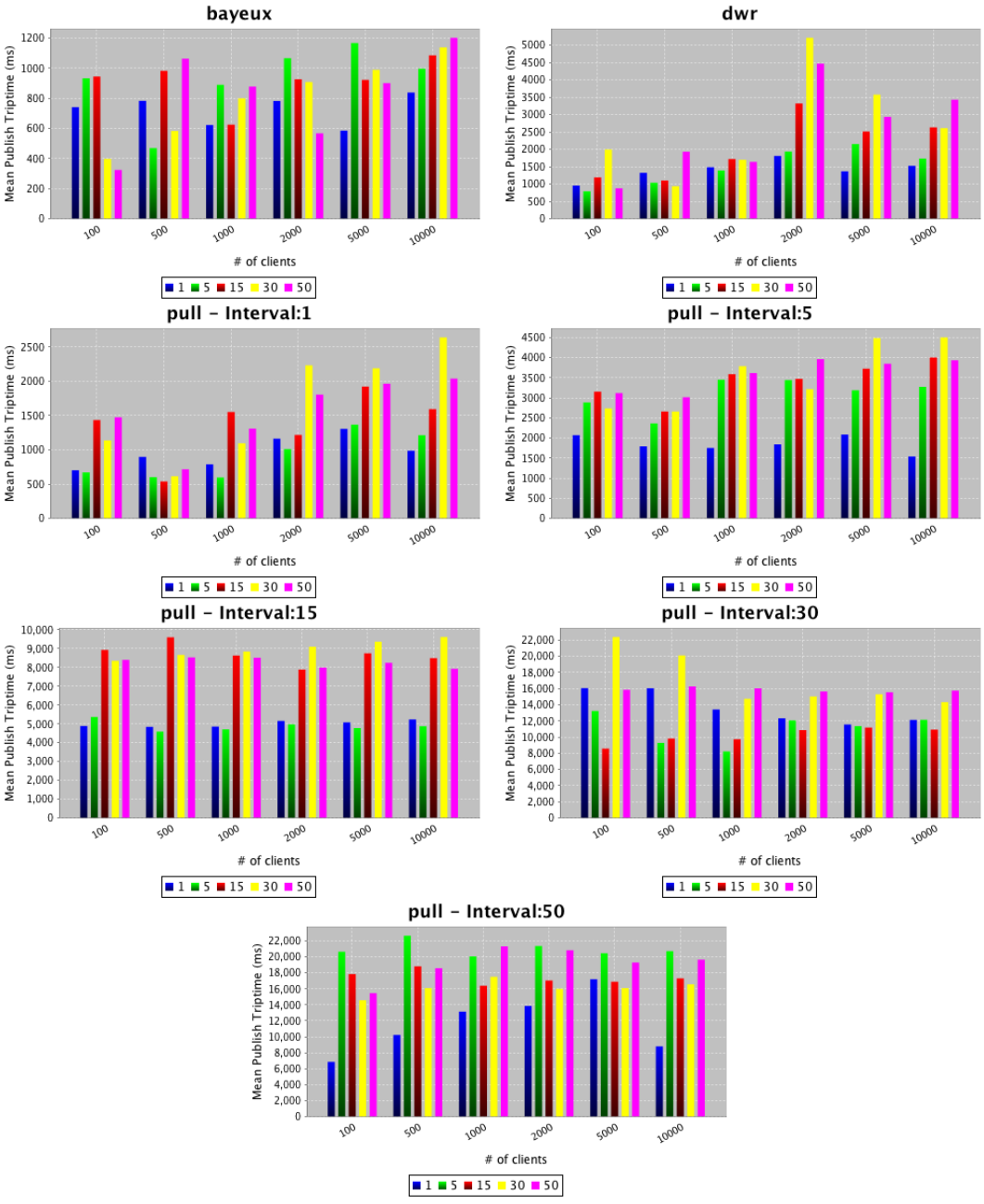

Fig. 9. Mean Publish Trip-time

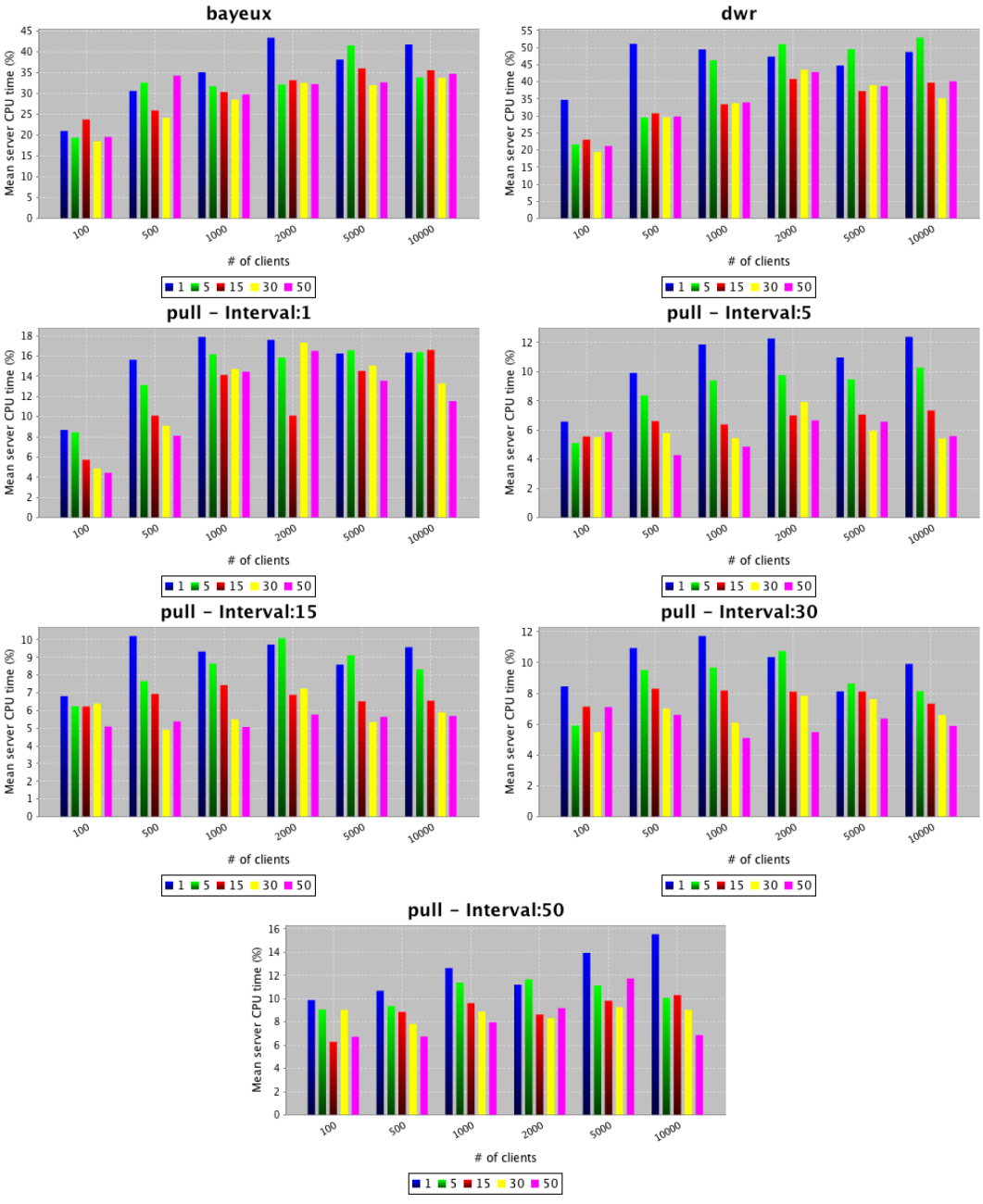

Fig. 10. Server application CPU usage.

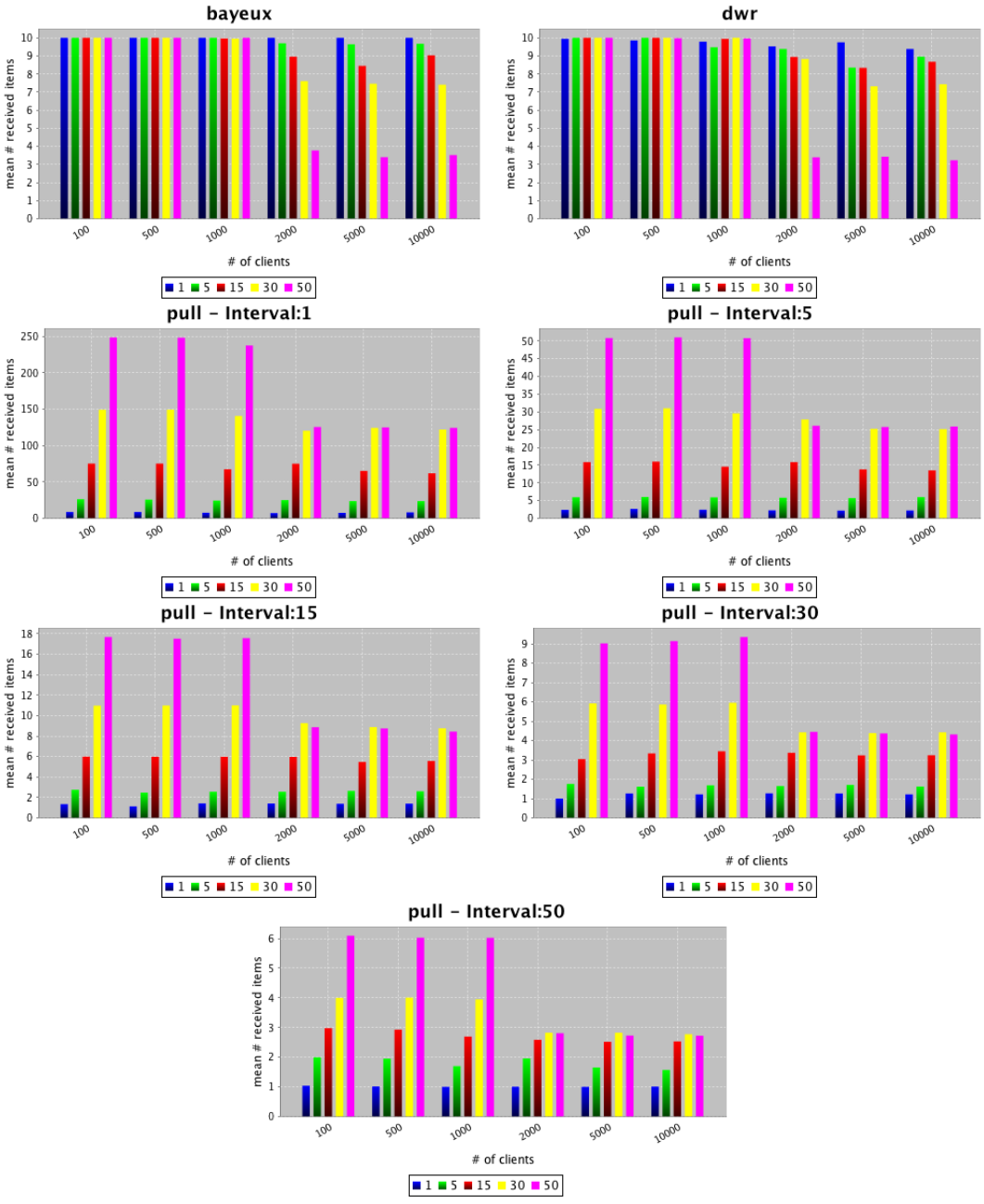

Fig. 11. Mean Number of Received Publish Messages

6.4 Received Unique Publish Messages

Figure 12 shows the mean number of received unique publish items versus total number of clients. Note that this shows the mean of Received Unique Publish Messages (RUPM) for only those clients that could make a connection to the server and receive data.

According to Figure 12, if the publish interval is higher or equal to the pull interval, the client will receive most of the messages. However as we have discussed in Section 6.3, this will generate an unnecessary number of messages. Looking at the figure again, we see that when the pull interval is lower than the publish interval, the clients will miss some updates, regardless of the number of users. So, with the pull approach, we need to know the *exact* publish interval. However, the publish interval tends to change, which makes it difficult for a pure pull implementation to adapt its pull interval.

With COMETD and DWR, almost all messages are received by the users (having up to a total of 2000 users). Higher than 2000 users, we see that some data miss begins to occur for both push libraries.

With pull, a lower pull interval leads to more accuracy. With a pull interval of 1, the mean number of received items can be as high as 9. With a pull interval of 5 this number drops to 6.

6.5 Received Message Percentage

Figure 13 shows us the Received Message Percentage (RMP). We notice that RMP is significantly higher with COMETD and DWR, compared to pull. We can also see that RMP decreases significantly in all approaches after 2000 users. This might depend on many factors, such as, Jetty application server queuing all the connections, or perhaps, before a client can process the test run ends. It might also lie within the client simulator, or our statistics server that have to cope with generating a high number of clients and receive a huge number of messages. See Section 7.2 for a discussion of possible factors that might have an effect on the results.

However, if we compare RMP of the three approaches for 2000 users or more, we see that COMETD and DWR clients still receive more data than pull, and thus have a higher degree of reliability.

6.6 Network Traffic

Figure 14 shows the results. We notice that, in COMETD and DWR, the number of TCP packets traveled to/from the server increases as the number of users increases, rising up to 80,000 packets with COMETD and 250,000 with DWR. We see that up to 2000 users, the publish interval has almost no effect, showing that the administrative costs of establishing a long polling connection is negligible. After 2000 users, there is a difference between different publish intervals, this might be caused by many factors including connection timeouts. With pull, in worst case scenario (`pullInterval =1 , publishInterval = 50, and 1000 users`), Network Traffic (NT) rises up to 550,000 packets. This is almost 7 times more than COMETD and 2 times more than DWR. This number decreases with lower pull intervals, however as we have discussed in Section 6.1, high pull intervals will lead to high trip-time and low data coherence. Also note that, with pull, NT increases as the publish interval increases. This is because a longer publish interval will lead to a longer test run, in turn leading to more pull requests.

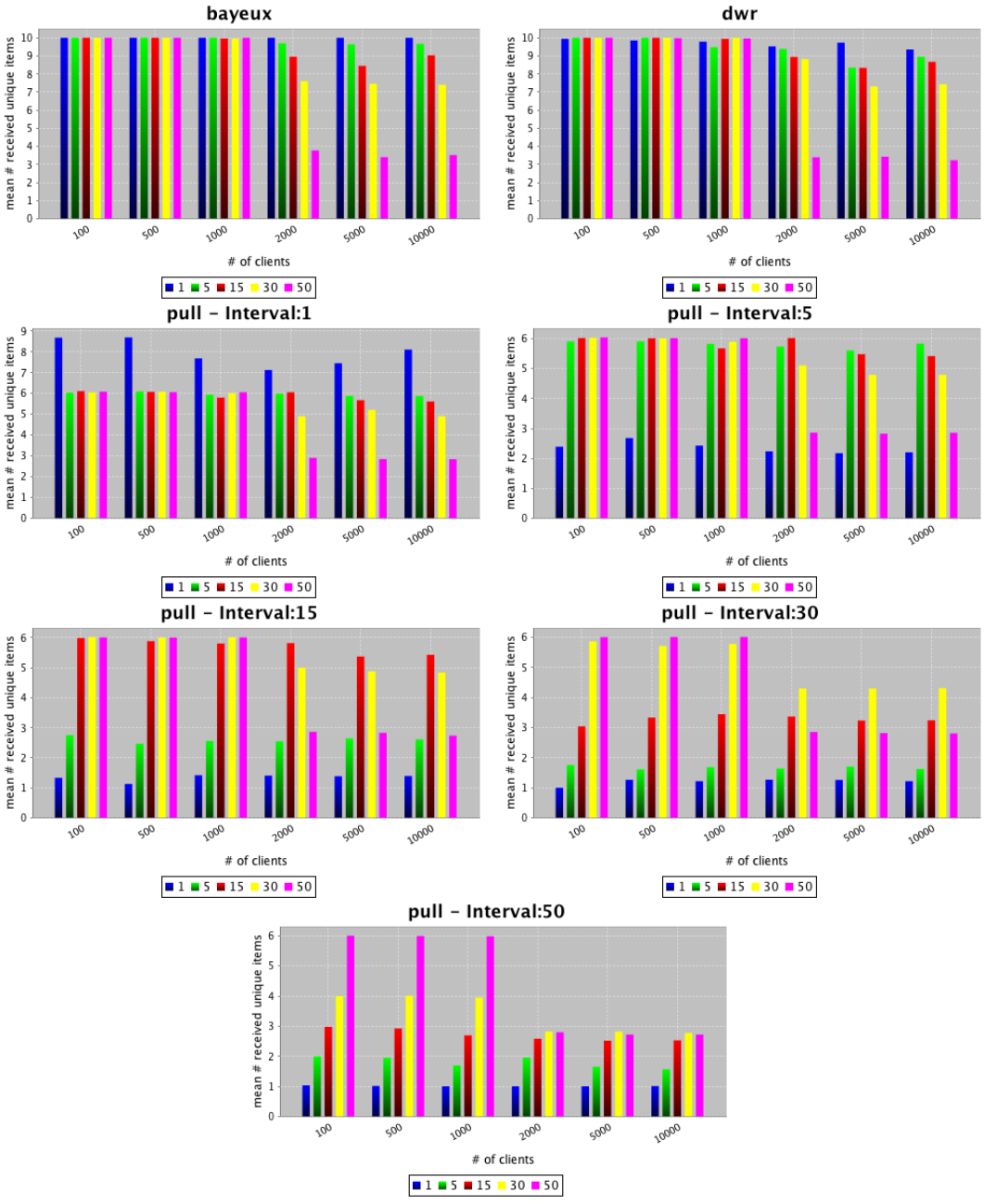

Fig. 12. Mean Number of Received Unique Publish Messages.

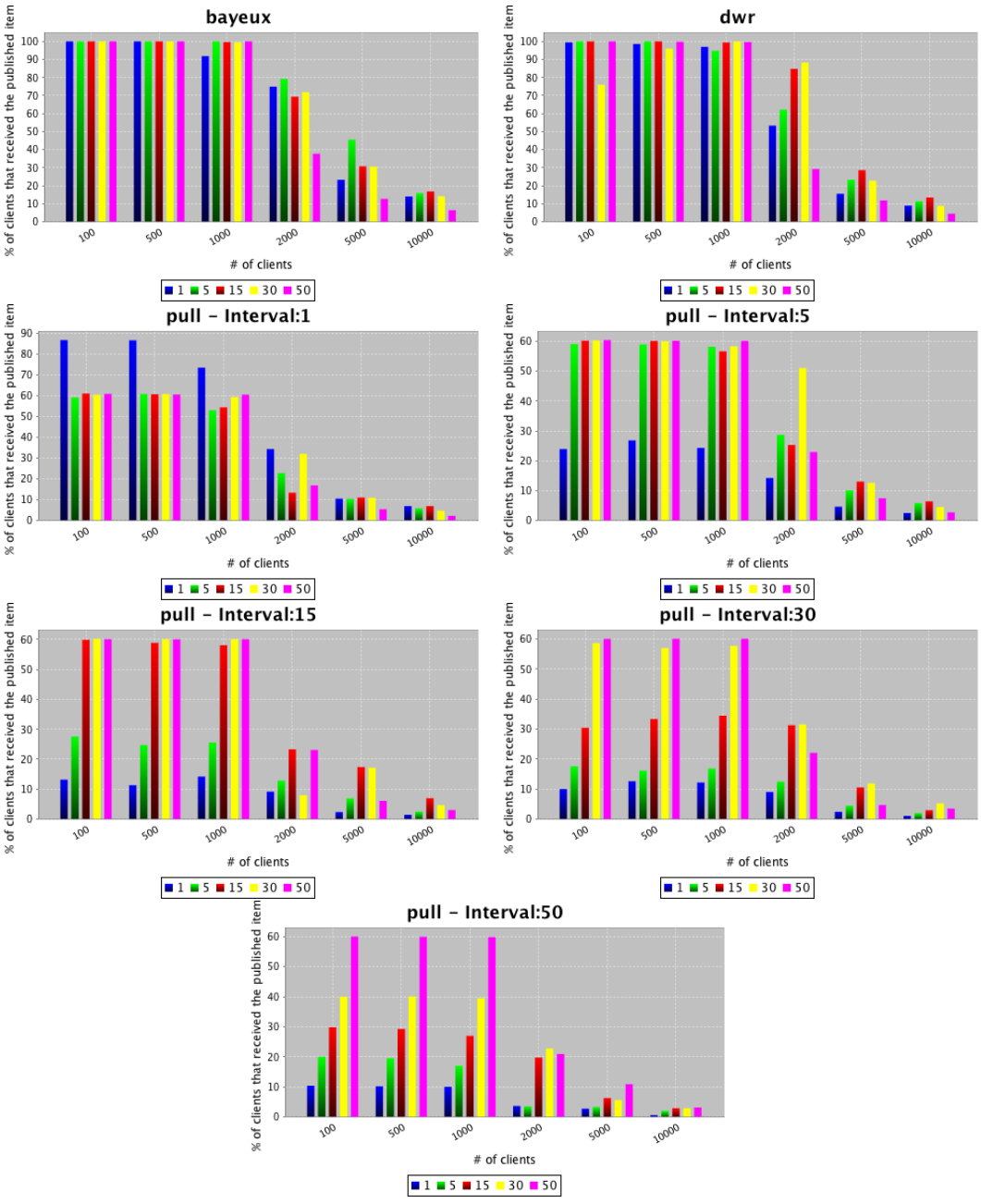

Fig. 13. Percentage of data items that are delivered to all clients

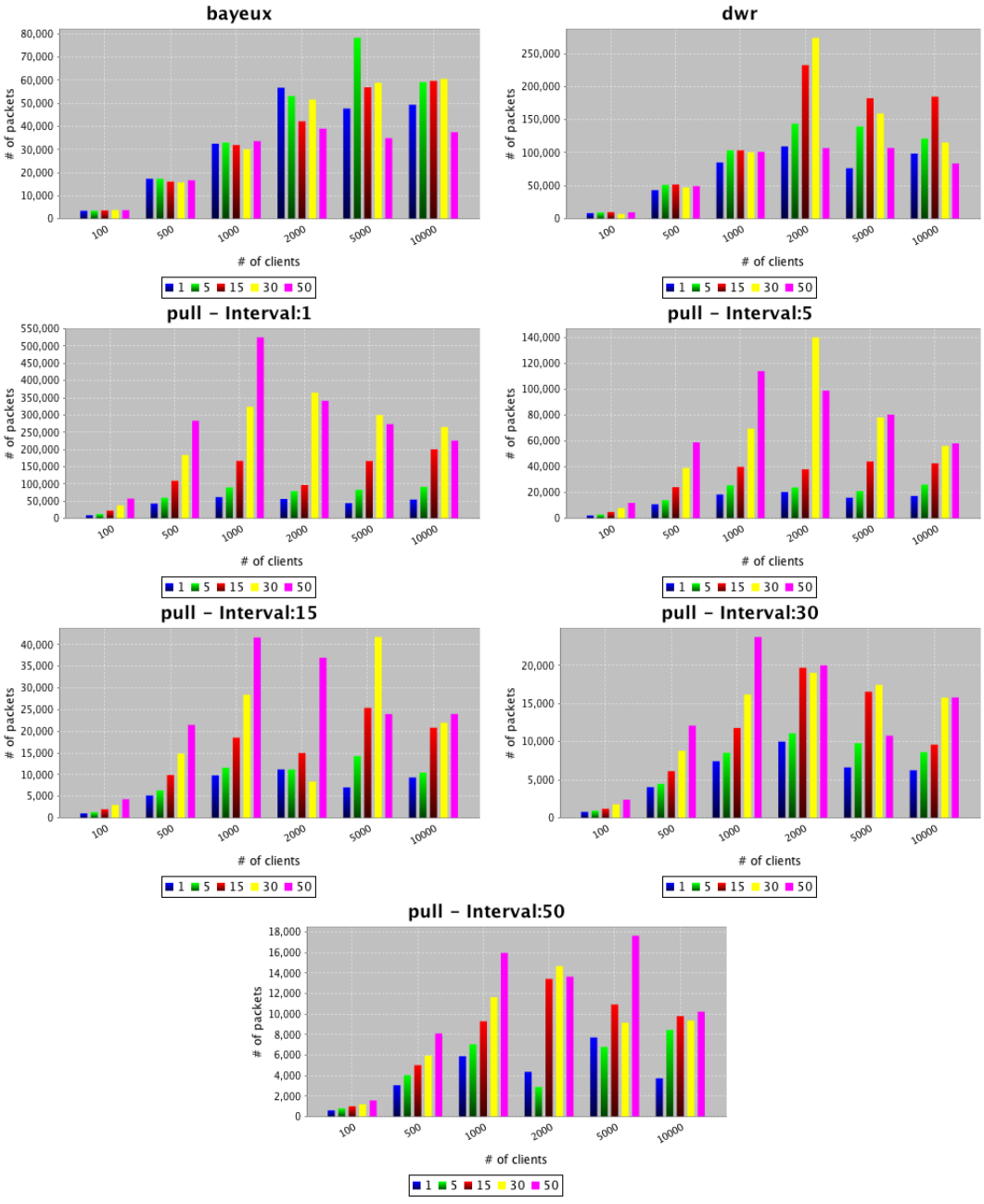

Fig. 14. Total number of TCP packets coming from/to the server

7 Discussion

In this section we discuss our findings and try to answer our research questions. We also present the threats to the validity of our experiment.

7.1 The Research Questions Revisited

RQ1 Data coherence: RQ1 inquired how fast the state changes (new messages) on the server were being propagated to the clients. We expected that the pull-based clients would have a higher trip-time value and thus a lower degree of data coherence. Our findings in Section 6.1 support our expectations. Even with a pull interval of 1 second, the trip-time with pull is higher than push. This finding indicates that the push-based style supports a higher degree of data coherence.

RQ2 Scalability and server performance: RQ2 addressed the issue with the scalability and the overall performance of the server. Considering our previous results [5], we initially expected push to cause significantly higher server load, compared to pull. However, our new findings in Section 6.2 show that, even though push causes more CPU usage, the difference is not significant. With push, the server CPU is not saturated, even with 10000 users. This indicates that both approaches scale well to thousands of clients.

RQ3 Network traffic: RQ3 tried to find to what extend the network traffic generated between the server and clients is influenced. We initially expected a smaller pull interval to create many unnecessary requests and a high network traffic. Our findings in Section 6.3 and Section 6.6 supported these predictions. Pull causes the clients to receive a higher number of redundant messages compared to push, leading to unnecessary network traffic. With pull, the number of traveling TCP packets to and from the server is 7 times more than COMETD.

RQ4 Reliability: RQ4 questioned the reliability of each approach. We expected that pull-based clients would miss many more data items compared to push clients. Our findings in Section 6.5 and Section 6.4 show that pull clients with a low pull interval are more up-to-date compared to those with a high pull interval, but also receive many redundant data if the publish interval is high. However, they are still missing many data items compared to the push-based clients.

7.2 Threats to Validity

A fundamental question concerning results from an experiment is how valid the results are [35]. In this section the validity of our findings is discussed with respect to *internal* and *external* validity. We structure our discussion according to Wohlin *et al.* [35], which in turn is based on the work of Campbell and Stanley [7], and Cook and Campbell [30].

7.2.1 Internal Validity

In an experiment, different treatments may be applied to the same object at different times. Then there is a risk that the history affects the experimental results, since the circumstances are not the same on both occasions [35]. We have the following threats to internal validity:

Non-deterministic nature of the distributed environments
Our experimental setup is distributed on different machines and each test run for each technique is run on different times of the day. A network congestion at the time of the test, might for example have an effect on the trip-time variable (See Section 6.1). We also use the supercomputer (DAS3) to

simulate many clients and at the time of testing there might be other users running tasks, which might affect the network bandwidth. In order to limit these effects on the network latency, we placed all the machines in the same LAN, used the same test-script in all the simulated clients and allocated the same bandwidth.

Reliability of the tools
We use several tools to obtain the result data. The shortcomings and the problems of the tools themselves can have an effect on the outcome. For example, from all the results we see that the performance suffers after 2000 users for all approaches. This degradation might be caused by the server, client simulator, or the statistics server. Debugging distributed applications, to find the cause of such behavior, proved to be very difficult.

Time and Data Coherence
The time calculation can also be a threat to the internal validity. To measure the trip-time, the difference between the data creation date and data receipt date is calculated. However if the time on the publisher and the clients is different, the trip-time is calculated incorrectly. In order to prevent this, we made sure that the time on the server and client machines are synchronized by using the same time server.

We measure the data coherence by taking the trip-time. However, the data itself must be correct, i.e., the received data must be the same data that was sent by the server. We rely on HTTP in order to achieve this data correctness. However, additional experiments must include a self check to ensure this requirement.

7.2.2 External validity

The external validity deals with the ability to generalize results [35]. There is a risk that the used *push* libraries are not good representatives of *push*, making it difficult to draw general conclusions about the whole *push* approach. In order to minimize the risk, we have used two libraries instead of one, and as we have shown in Section 6, we see the same pattern, which confirms our findings for push. However, we did only use a single web application server, because at the time of the testing Jetty was the only open source and stable Java server that supported NIO. In the future, different application servers should be included, e.g., Sun's Grizzly[x].

We only used one type of sample application, namely the stock ticker. In this scenario, we had a single channel with many users, where a data item is sent to all the push clients. However, there are other use cases, such as a chat application, where there will be multiple channels with many users. In this scenario, a data item will only be sent to the subscribers of that particular channel. This will have effects on the scalability and other dependent variables. Therefore, further tests with different use cases are necessary.

In order to limit the external factors that affect the trip-time, we placed all the test machines in the same network and all the users are granted with the same bandwidth. In a real-life scenario, users are located at different parts of the world, and have different bandwidth properties, leading to a bigger variance in the trip-time. This should be taken into account before deciding on actual parameters of a web application using *push*.

During the experiment execution, if a large volume of data exchange occurs, this might lead to concurrency on the access to the shared resources. To minimize this threat, we run each server (ap-

[x] https://grizzly.dev.java.net/

plication server, client generator, statistics server) on different machines. Note that only the client generator is located in a cluster (DAS3). Other servers are located outside the cluster.

8 Related Work

There are a number of papers that discuss server-initiated events, known as *push*, however, most of them focus on client/server distributed systems and non HTTP multimedia streaming or multi-casting with a single publisher [1, 16, 13, 3, 31]. The only work that focuses on AJAX is the white-paper of Khare [17]. Khare discusses the limits of the pull approach for certain AJAX applications and mentions several use cases where a push application is much more suited. However, the white-paper does not mention possible issues with this *push* approach such as scalability and performance. Khare and Taylor [18] propose a push approach called ARRESTED. Their asynchronous extension of REST, called A+REST, allows the server to broadcast notifications of its state changes. The authors note that this is a significant implementation challenge across the public Internet.

The research of Acharya *et al.* [1] focuses on finding a balance between push and pull by investigating techniques that can enhance the performance and scalability of the system. According to the research, if the server is lightly loaded, pull seems to be the best strategy. In this case, all requests get queued and are serviced much faster than the average latency of publishing. The study is not focused on HTTP.

Bhide *et al.* [4] also try to find a balance between push and pull, and present two dynamic adaptive algorithms: *Push and Pull* (PaP), and *Push or Pull* (PoP). According to their results, both algorithms perform better than pure pull or push approaches. Even though they use HTTP as messaging protocol, they use custom proxies, clients, and servers. They do not address the limitations of browsers nor do they perform load testing with high number of users.

Hauswirth and Jazayeri [15] introduce a component and communication model for push systems. They identify components used in most *Publish/Subscribe* implementations. The paper mentions possible problems with scalability, and emphasizes the necessity of a specialized, distributed, broadcasting infrastructure.

Eugster *et al.* [11] compare many variants of *Publish/Subscribe* schemes. They identify three alternatives: *topic-based*, *content-based*, and *type-based*. The paper also mentions several implementation issues, such as events, transmission media and qualities of service, but again the main focus is not on web-based applications.

Flatin [19] compares push and pull from the perspective of network management. The paper mentions the publish/subscribe paradigm and how it can be used to conserve network bandwidth as well as CPU time on the management station. Flatin suggests the 'dynamic document' solution of Netscape [24], but also a 'position swapping' approach in which each party can both act as a client and a server. This solution, however, is not applicable to web browsers. Making a browser act like a server is not trivial and it induces security issues.

As far as we know, there has been no empirical study conducted to find out the actual trade-offs of applying pull/push on browser-based or AJAX applications.

9 Conclusion

In this paper we have compared pull and push solutions for achieving web-based real time event notification and data delivery. The contributions of this paper include:

- An experimental design permitting the analysis of pull and push approaches to web data delivery, and the identification of key metrics, such as the mean publish trip-time, received (unique) publish messages, and the received message percentage (Section 4).
- A reusable software infrastructure, consisting of our automated distributed AJAX performance testing framework CHIRON, Grinder scripts imitating clients and a sample application written for COMETD, DWR, and *pull* (Section 5).
- Empirical results highlighting the tradeoffs between push and pull based approaches to web-based real time event notification, and the impact of such characteristics as the number of concurrent users, the publish interval, on, for instance, server performance (Section 6).

Our experiment shows that if we want high data coherence and high network performance, we should choose the push approach. Pull cannot achieve the same data coherence, even with low pull intervals. Push can also handle a high number of clients thanks to the continuations [9] mechanism of Jetty, however, when the number of users increases, the reliability in receiving messages decreases.

With the pull approach, achieving total data coherence with high network performance is very difficult. If the pull interval is higher than the publish interval, some data miss will occur. If it is lower, then the network performance will suffer, in some cases pull causes as high as 7 times more network traffic compared to push. Pull performs close to push only if the pull interval equals to publish interval, but never better. Besides, in order to have pull and publish intervals equal, we need to know the exact publish interval beforehand. The publish interval on the other hand is rarely static and predictable. This makes pull useful only in situations where the data is published according to some pattern.

These results allow web engineers to make rational decisions concerning key parameters such as pull and push intervals, in relation to, e.g., the anticipated number of clients. Furthermore, the experimental design and the reusable software infrastructure allows them to repeat similar measurements for their own (existing or to be developed) applications. We will soon release CHIRON through our website (See Section 5).

Our future work includes adopting and testing a hybrid approach that combines pull and push techniques for AJAX applications to gain the benefits of both approaches. In this approach for example, users can specify a maximum trip-time, and if the server is under high load, it can switch some push users to pull. We believe that such optimizations can have a positive effect on the overall performance. We also intend to extend our testing experiments with different web application containers such as Grizzly, or different push server implementations that are based on holding a permanent connection (e.g., Lightstreamer[y]) as opposed to the long polling approach discussed in this paper.

References

[1] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcast. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 183–194. ACM Press, 1997.

[2] S. Alager and S. Venkatsean. Hierarchy in testing distributed programs. In *AADEBUG '93: Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, pages 101–116, London, UK, 1993. Springer-Verlag.

[y] http://www.lightstreamer.com

[3] M. Ammar, K. Almeroth, R. Clark, and Z. Fei. Multicast delivery of web pages or how to make web servers pushy. Workshop on Internet Server Performance, 1998.

[4] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: Disseminating dynamic web data. *IEEE Trans. Comput.*, 51(6):652–668, 2002.

[5] E. Bozdag, A. Mesbah, and A. van Deursen. A comparison of push and pull techniques for Ajax. In S. Uang and M. D. Penta, editors, *Proc. of 9th IEEE International Symposium on Web Site Evolution (WSE)*, pages 15–22, 2007.

[6] L. C. Briand, S. Morasca, and V. R. Basili. An operational process for goal-driven definition of measures. *IEEE Trans. Softw. Eng.*, 28(12):1106–1125, 2002.

[7] D. Campbell and J. Stanley. *Experimental and Quasi-Experimental Designs for Research*. Rand-McNally, Chicago, 1963.

[8] J. Chen, R. M. Hierons, and H. Ural. Overcoming observability problems in distributed test architectures. *Inf. Process. Lett.*, 98(5):177–182, 2006.

[9] M. Consulting. Jetty webserver documentation - continuations. `http://docs.codehaus.org/display/JETTY/Continuations`, 2007.

[10] Direct Web Remoting. Reverse Ajax documentation. `http://getahead.org/dwr/reverse-ajax`, 2007.

[11] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[12] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Trans. Inter. Tech.*, 2(2):115–150, 2002.

[13] M. Franklin and S. Zdonik. Data in your face: push technology in perspective. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 516–519. ACM Press, 1998.

[14] J. Garrett. Ajax: A new approach to web applications. Adaptive Path: `http://adaptivepath.com/publications/essays/archives/000385.php`, 2005.

[15] M. Hauswirth and M. Jazayeri. A component and communication model for push systems. In *ESEC/FSE '99*, pages 20–38. Springer-Verlag, 1999.

[16] K. Juvva and R. Rajkumar. A real-time push-pull communications model for distributed real-time and multimedia systems. Technical Report CMU-CS-99-107, School of Computer Science, Carnegie Mellon University, January 1999.

[17] R. Khare. Beyond Ajax: Accelerating web applications with real-time event notification. Knownow.com, white-paper.

[18] R. Khare and R. N. Taylor. Extending the representational state transfer (REST) architectural style for decentralized systems. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 428–437. IEEE Computer Society, 2004.

[19] J.-P. Martin-Flatin. Push vs. pull in web-based network management. `http://arxiv.org/pdf/cs/9811027`, 1999.

[20] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling Ajax by inferring user interface state changes. In *Proceedings of the 8th International Conference on Web Engineering (ICWE'08)*, pages 122–134. IEEE Computer Society, 2008.

[21] A. Mesbah and A. van Deursen. An architectural style for Ajax. In *Proc. of 6th Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, pages 44–53, 2007.

[22] A. Mesbah and A. van Deursen. Migrating multi-page web applications to single-page Ajax interfaces. In *Proc. of 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 181–190, 2007.

[23] A. Mesbah and A. van Deursen. A component- and push-based architectural style for Ajax applications. *Journal of Systems and Software (JSS)*, 2008. To appear.

[24] Netscape. An exploration of dynamic documents. `http://wp.netscape.com/assist/net_sites/pushpull.html`, 1996.

[25] A. Russell. Comet: Low latency data for the browser. `http://alex.dojotoolkit.org/?p=545`.

[26] A. Russell, G. Wilkins, and D. Davis. Bayeux - a JSON protocol for publish/subscribe event delivery protocol 1.0draft1. `http://svn.xantus.org/shortbus/trunk/bayeux/bayeux.html`, 2007.

[27] D. Schiemann. The forever-frame technique. `http://cometdaily.com/2007/11/05/the-forever-frame-technique`, November 2007.

[28] R. Srinivasan, C. Liang, and K. Ramamritham. Maintaining temporal coherency of virtual data warehouses. In *RTSS '98: Proc. of IEEE Real-Time Systems Symposium*, page 60, 1998.

[29] Sun microsystems remote method invocation home. `http://java.sun.com/javase/technologies/core/basic/rmi/`.

[30] T.D.Cook and D.T.Campbell. *Quasi-Experimentation - Design and Analysis Issues for Field Settings*. Houghton Mifflin Company, 1979.

[31] V. Trecordi and G. Verticale. An architecture for effective push/pull web surfing. In *2000 IEEE International Conference on Communications*, volume 2, pages 1159–1163, 2000.

[32] W3C. Chunked transfer coding. `http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html#sec3.6.1`.

[33] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.

[34] M. Welsh and D. E. Culler. Adaptive overload control for busy internet servers. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[35] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, 2000.