Evaluation of Long-Held HTTP Polling for PHP/MySQL Architecture

David Cutting

University of East Anglia — Purplepixie Systems
David.Cutting@uea.ac.uk | dcutting@purplepixie.org

Abstract. When a web client needs to periodically refresh data held on the server there are generally two approaches. Interval polling ("short polling"), which is most commonly used, where the client repeatedly reconnects to the server for updates, and a technique in which the HTTP connection is kept open ("long polling"). Although work exists investigating the possibilities of long polling few if any experiments have been performed using an Apache MySQL PHP (AMP) stack. To determine the potential effectiveness of long polling with this architecture an experiment was designed and conducted to compare update response times using both methods with a variety of polling intervals. Results clearly show a marked improvement in timings with long polling, with the mean response time down to 0.38s compared to a mean of just under the polling interval for short polling. Issues of the complexity and load implications of wide use of long polling are discussed but were outside the remit of this experiment owing to a lack of resources.

Keywords: HTTP, PHP, MySQL, Long-Held Polling, long poll, short poll, interval polling

1 Introduction

Within web systems there are commonly two alternative methods used to meet a requirement for client updates, typical interval based polling and long-held HTTP requests. Although work has been done in this area most deals specifically with Java as the server-side technology so in order to establish the feasibility of the less-common long-held method and evaluate relative performance in PHP an experiment was proposed. This experiment formed part of the development of an open-source extensible service desk system (FreeDESK) using Apache MySQL PHP (AMP) architecture [Cutting, 2012].

1.1 Background

The Hypertext Transfer Protocol (HTTP) is a protocol built around the request/response paradigm whereby a client establishes a connection to a server, makes a request, receives a response and the connection is then closed [Loreto et al., 2011].

As the server cannot therefore make connections to the client, for example to notify it of events, the client must repeatedly poll the server for updates, an inefficient method known as "short polling" involving significant overhead as new connections are set up then torn down for each poll and client updates can only be received once per polling interval [Loreto et al., 2011].

Short polling occurs "blindly" irrespective of a data update being present and in order for low latency (high data/display accuracy) the polling interval must be low meaning a high use of resources [Bozdag et al., 2007].

One approach to resolve this issue is the use of long-held HTTP polling ("long polling") a technique where the connection between the client and server is held open until data is available and sent [Loreto et al., 2011]. Current examples of this technology in use include Google products such as gMail and gTalk [Russell, 2006].

As many modern web-based systems (including the FreeDESK system) will make use of asynchronous Javascript XML requests (AJAX) and will need a mechanism to ensure screen updates are picked up and reflected within the user interface, consideration must be given as to how this is accomplished; through traditional short polling or using a held open connection with long polling. To examine the difference in response times between these techniques an experiment was performed.

Although Bozdag et al. [Bozdag et al., 2007] presents a similar experiment this was constructed using Java as a base technology and also a messaging bus. The architecture of the FreeDESK system being PHP and a relational SQL database has a different structure so this experiment is necessary to check performance against the specific FreeDESK architecture or any other system based on AMP.

Long polling offers potential benefits with latency but also has inherent issues including the server overhead from connections being held open, timeout problems where the connection is timed out by the client and/or server and caching where request responses are cached in an intermediate proxy or web cache and then returned time and again [Loreto et al., 2011].

There can be no doubt that the issue of server resources is serious especially in large or shared deployments where many clients are using the same HTTP server. Owing to a lack of resources and in order to keep the experiment focused consideration of server resources are outside the scope of this experiment. This was investigated by Bozdag et. al [Bozdag et al., 2007] who did produce some quantifiable results with regard to server loading.

In this experiment timeout problems were mitigated with a built-in drop and reconnect period whereby, after a specified period (25 seconds), before the browser request or PHP script will themselves timeout, the connection is closed with no data, causing the client to perform an immediate reconnect. To avoid caching conflicts all data will be sent and received using HTTP POST rather than HTTP GET which should not be cached by intermediate proxies. As an additional safeguard a "nocache" field will also be passed containing a randomised (and therefore different) string for each request which, though ignored by the

server, will mean the POST request contains different data each time and so should definitely not be cached. Data returned from the server will have headers marked to show it should not be cached and has "expired" already.

2 Hypothesis

Before conducting the experiment the following hypothesis were made both the expected outcome (H_1) and also a null hypothesis (H_0).

H_1: Long polling will offer significant reductions in latency and bring average update response time to 0.5s or less

H_0: Long polling will show no significant reductions in latency over short polling

3 Experiment Design

To replicate the architecture of a production AMP system (such as FreeDESK) as much as possible the experiment was designed with a PHP front-end and an underlying SQL database. When a test session was in progress it would move through stages of short polling (with 10 and 5 second polling intervals) and long polling. A series of messages would be generated at randomised intervals in the database and then provided to the test client which would then acknowledge their receipt allowing the server to calculate a total "acknowledgement time" (the time from the creation of the message to the acknowledgement being received from the client).

Figure 1 illustrates a timeline using short polling to a system with an underlying SQL database and figure 2 shows the timeline in a long polling context.

In order to detect the effects of general network latency intermediate stages on the client performed a "static fetch" where a small (65 byte) static XML file was be fetched from the server with the client recording the time taken from request to response and then passing this information to the server for inclusion in the overall dataset. This allowed for identification of sessions with unusually high general latencies that could either be removed from the dataset or adjusted accordingly giving a truer estimate of latencies in the fetching only.

Stage	**Client**	**Server**
0 Registration	Generate randomised session ID and register with server	Register session and spawn message generation thread
1 Static Fetch	Perform three fetches of static content and record times	
2 Short Poll (10s)	Perform periodic (10s) poll of server for new messages, acknowledge receipt immediately	Check for new messages in SQL database and provide them, record acknowledgement times

3 Static Fetch	As 1	
4 Short Poll (5s)	As 2 with 5s period	As 2
5 Static Fetch	As 1	
6 Long Poll	Perform long polling, recycling a new connection on every response with acknowledgement of any data received	Keep connection alive and periodically (500ms) poll the SQL database for new messages, recycling after 25s, record acknowledgement times
7 Static Fetch	As 1	
8 Complete	Send completion message including static fetch times	Mark session as complete (so message thread can exit), move message data into completed table, record static fetch times

Table 1: Long-Held HTTP Experiment Stages

The experiment was written in PHP and SQL for the server-side components and HTML with Javascript for the client-side. The full experiment source code can be downloaded from `http://www.purplepixie.org/davestuff/sleepjax.zip`.

4 Experiment

The experiment was conducted over two weeks in early July 2012. The server installation was on a Linux based server running Apache 2 located in a data centre in Manchester, England[1]. Clients connected from numerous locations in the United Kingdom using a number of different web browsers, platforms and connections. This ranged from Internet Explorer under Windows on a fixed broadband line to Android Browser on a mobile phone using GPRS/3G. The experiment running on a Samsung Galaxy SIII phone can be seen in figure 3.

[1] Operated by Melbourne Server Hosting `http://www.melbourne.co.uk/`

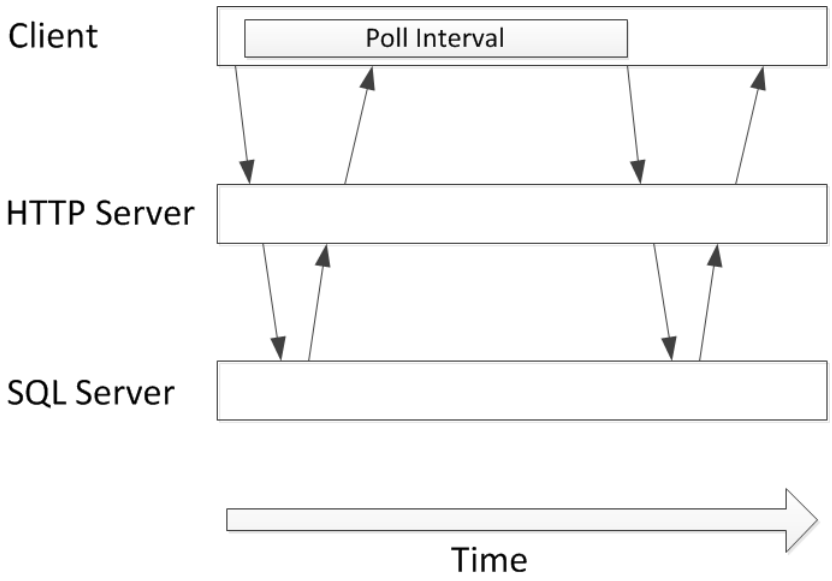

Fig. 1. Short Polling to HTTP Server with Underlying SQL Database (not to scale)

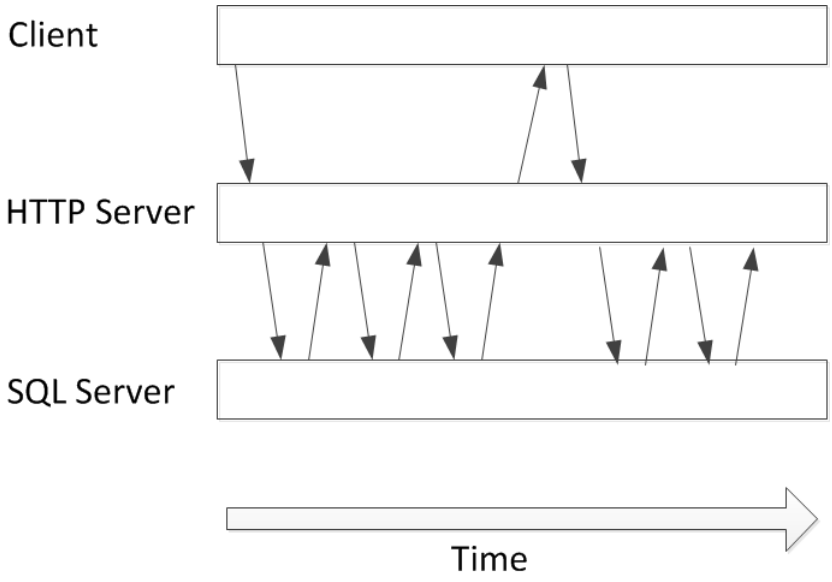

Fig. 2. Long Polling to HTTP Server with Underlying SQL Database (not to scale)

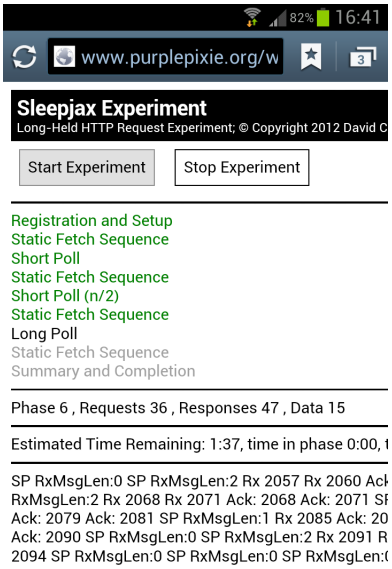

Fig. 3. Experiment Running in Android Browser on a Phone using WiFi

5 Results

Test Sessions	101
Messages Passed	2238
10s Short Poll Acknowledgement	4.8691s Mean $\sigma = 2.997$
5s Short Poll Acknowledgement	2.3915s Mean $\sigma = 1.429$
Long Poll Acknowledgement	0.3812s Mean $\sigma = 1.429$
Static Fetch	0.0563s Mean $\sigma = 0.065$

Table 2: Experiment Headline Results

Figure 4 shows the distribution of response times using the different polling methods to the nearest 0.1s.

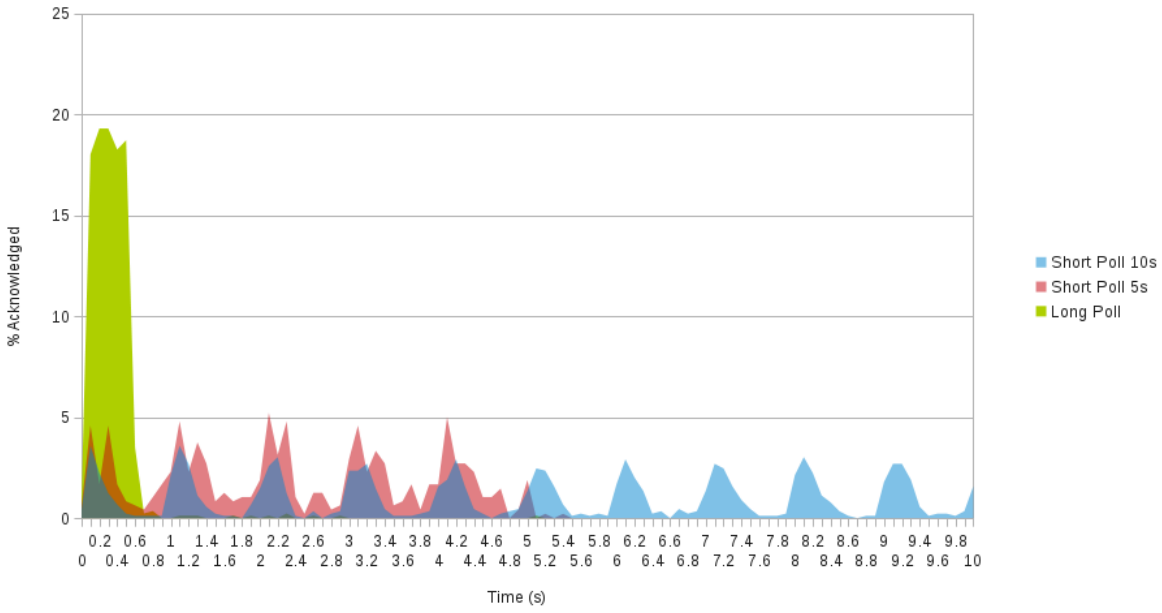

Fig. 4. Experiment Acknowledgment Times to Nearest 0.1s

6 Conclusion

From the results it can clearly be seen that the use of long polling provides a significant reduction in latency with the mean acknowledgement time for a data message dropping from 4.7s and 2.4s respectively for 10 and 5 second interval short polling to a 0.4s for long polling.

Not only was the long poll mean under 0.5s but 85.27% (n=735) of long poll messages were acknowledged in 0.5s or under. This data therefore strongly supports the hypothesis (H_1) and the null hypothesis (H_0) can be discarded.

A portion of the experiment was to repeat a series of "static fetches" and record results to allow for the identification of test sessions where high overall network latency was a factor. It was found that, even with a large range of connection methods ranging from corporate leased lines to GRPS, the static times were consistent with a mean of 0.07s and a standard deviation of only 0.065. For this reason analysis of results against and considering static transfer times was not completed.

An area of consideration not covered in this experiment but analysed in Bozdag et al. [Bozdag et al., 2007] was server resource utilisation. An additional resource constraint in the AMP context is that of repeated SQL database polling by the HTTP server to find new data. Bozdag et al. [Bozdag et al., 2007] showed that although the use of long polling in their experiment also offered a significant reduction in latency times server load was greatly increased especially when handling a large number of clients.

Unfortunately given resource constraints this experiment could not replicate the effects of large numbers of clients connecting but it is plausible that server load would increase inline or even ahead of the Bozdag et al. [Bozdag et al., 2007] findings.

Additionally the architecture of a system to support long polling must be necessarily more complex than one for short polling. Rather than allowing a simple "refresh" of data after a polling intervals changes must be detected and then passed to the client as and when they occur. This requires specific handling at all levels of the system and the ability to record and identify changes to data against an existing view of that data, potentially a complex and resource intensive task.

For these reasons it is concluded that long polling can offer significant benefits in latency and update speed but at the risk of much higher complexity and load. Certain specific applications demanding near real-time updates but with few clients, or one in which the server capacity can be easily scaled without inherent complexity and within cost limits may be very applicable to long polling, but most generalised applications suitable for a wide range of deployment configurations are not easily applicable.

Further work should include a more detailed analysis of the server load profiles against differing numbers of clients and complexities of systems to build a broader view of the relative costs and benefits of using long polling.

Notes

This paper is a technical working paper primarily containing the findings already contained in [Cutting, 2012], just published separately for clarity. The author is affiliated to both Purplepixie Systems and the University of East Anglia. Published November 2015.

References

[Bozdag et al., 2007] Bozdag, E., Mesbah, A., and Van Deursen, A. (2007). A comparison of push and pull techniques for ajax. In *Web Site Evolution, 2007. WSE 2007. 9th IEEE International Workshop on*, pages 15–22. IEEE.

[Cutting, 2012] Cutting, D. (2012). Free open-source extensible helpdesk (freedesk). Masters dissertation, University of East Anglia. http://freedesk.purplepixie.org/FreeDESK-Dissertation-FDL.pdf.

[Loreto et al., 2011] Loreto, S., Saint-Andre, P., Salsano, S., and Wilkins, G. (2011). Rfc 6202: Known issues and best practices for the use of long polling and streaming in bidirectional http. `http://www.ietf.org/rfc/rfc6202.txt`. [Online; accessed July 2012].

[Russell, 2006] Russell, A. (2006). Comet: Low latency data for browsers. *alex. dojotoolkit. org*.