A scalability study into Server Push technologies with regard to server performance

Final Report

Student Name:	John Frizelle
Student Number:	96015268
Submission Date:	01 September 2011
Supervisor:	Dr. Kieran Murphy

Table of Contents

Table of Figures

1. Introduction

1.1 Background

The nature of the Internet has changed dramatically since the first World Wide Web Consortium (W3C) HTTP [1] protocol specifications in the mid-1990s. The original HTTP specification was a simple Request-Response model whereby a client (browser) issued a request for a resource (web page, image, data etc.) and the server responded with the required data.

In the last few years, new momentum has been gathering behind a technology which has existed in various guises for many years. This technology is known by various names including Server Push, Long Polling, Reverse Ajax and Comet. While each of these technologies have some implementation differences, they all share the same common ideal – to develop a means of overcoming the limitations of the traditional Request-Response model.

The goal of these technologies is to allow the server to send information to the client in real time without first having to wait for the client to issue a request. In reality, due to the nature of the HTTP protocol, there is still a requirement for a client request. However, with this family of technologies, when a request is received, the server does not respond immediately if there is no data available to send. Instead, the server holds the connection open until such time as data becomes available. At this point the server "pushes" the data to the client.

This new suite of technologies (which I will refer to in general terms as Server Push) is not without its drawbacks:

1. Server Architecture – Depending on the server architecture in use, long running requests can have a high overhead associated with them and so may not scale well.
2. Connection Duration – There are challenges with regard to how long a HTTP connection can be kept open without any data being sent before the client, the server, or some intermediary proxy considers the request dead and closes the HTTP connection.

This project aims to review two specific Server Push technologies (Long Poll and Comet) with a focus on how they affect server performance and how various server architectures can be modified to improve their performance.

1.2 Server Push Technologies

1.2.1 Long Polling

With the long poling approach, the client-initiated connection is kept open until either:

1. Data is available to send to the client.
2. A pre-defined poll timeout is exceeded.

After either of these two events occurs the connection is closed. At this point the client will issue another Long Polling request.

1.2.2 Comet

The Comet approach is to keep the connection open indefinitely. Data can be sent to the client as it becomes available, and the connection is not closed once data has been sent. The only time the client will re-issue a request is if the connection is dropped or closed due to an error or external timeout.

1.3 Document Outline

This document provides a detailed investigation of the performance characteristics of Long Poll and Comet style Server Push technologies. This document does not seek to provide a broad background for the project being undertaken as that information was presented in the Project Proposal document which was submitted in January 2011. Nor does it seek to provide a detailed review of academic literature in this field, as that information was presented in the Literature Review document which was submitted in June 2011.

The following sections are presented in this document:

Section 2:	Literature Review: A high level outline of the relevant literature which has been reviewed to date.
Section 3:	System Overview: A detailed insight into the various components of the systems developed in order to perform the various Server Push tests.
Section 4:	Test Preparation and Permutations: A description of the steps taken to ensure that the systems under test were configured for optimal performance prior to testing. Details of the various permutations of test which were carried out and how these were combined to identify the final test suite.
Section 5:	Results and Findings: An exploration of the tests executed and the results gathered. Results are presented from various view points.
Section 6	Conclusion and Future Work: Provides details of the conclusions drawn based on the findings from section 5 and how these findings relate to the research questions which were originally proposed. It also highlights some potential areas for future work.

2. Literature Review

This section provides a brief synopsis of literature reviewed in preparation for the development of the test system and execution of tests. For a more in-depth literature review, please refer to the Literature Review document submitted in June 2011.

2.1 Server Push Technologies

The term Comet was first coined by Alex Russell [2] of the DoJo Toolkit project. When comparing Comet and Long Polling, he made the following observations:

> Polling frees resources quickly, but makes many times as many requests… Polling is a latency/load trade-off.

In effect what Russell was referring to here is the need to get the balance right between the volume and duration of Long Polling requests.

With regard to the Comet approach, Russell makes the following observations:

> Comet is an architectural complexity trade-off (today). Most of today's web servers use threads or processes. Threads consume fixed resources per request… Do not free them until end of connection. Comet does not free connections quickly.

With most current thread based servers, this can quickly lead to thread starvation resulting in one of two outcomes:

1. Server Crash: If the number of threads is set too high, the server will not be able to cope with the volume of active threads.
2. Refused Connections: Once all available connections are assigned to Comet data transfer, new requests cannot be serviced.

2.2 Server Architecture – Threads vs. Events

Thread based and event based servers take a fundamentally different approach to handling concurrent requests. M. McGranaghan captures this fundamental difference very clearly in his "Threaded vs Evented Servers" article [3]:

> Threaded servers use multiple concurrently-executing threads that each handle one client request, while evented servers run a single event loop that handles events for all connected clients.

As noted by Von Behren et al [4], the debate between thread vs event based architecture is a very old one. In their article, they cite a 1978 paper from Lauer and Needham [5] who put forth the argument that "Procedure-oriented System" (i.e. thread based) and "Message-oriented System" (i.e. event based) are in fact

> …duals of each other and that a system which is constructed according to one model has a direct counterpart in the other. The principal conclusion is that neither model is inherently preferable.

Von Behren et al disagree with Lauer and Needham conclusions, arguing that:

> …the simpler programming model and wealth of compiler analyses that threaded systems afford gives threads an important advantage over events when writing highly concurrent servers.

J Ousterhour [6] highlights the following issues with threads:

1. Synchronization: Must coordinate access to shared data with locks
2. Hard to debug: data dependencies, timing dependencies
3. Threads break abstraction: can't design modules independently

However, he then goes on to say the following of events:

1. Long-running handlers make application non- responsive
2. Can't maintain local state across events
3. No CPU concurrency

The summary advice from Ousterhour is:

> …avoid threads wherever possible: Use events, not threads, for GUIs, distributed systems, low-end servers. Only use threads where true CPU concurrency is needed.

2.3 Performance Testing

The goal of performance testing is to measure how a system will perform when subjected to a high volume of concurrent connections. As noted by Repp et al [8]

> According to the W3C, performance is defined in terms of throughput, reponse [sic] time, latency, execution time, and transaction time.

Performance testing is not a new concept; as early as 1971, H. Lucas Jr of Stanford University [9] was producing academic literature in this field.

Traditional performance testing typically focuses on the definition of performance requirements for specific systems. For example, Weyuker and Vokolos provide the following example of a performance requirement:

> A more satisfactory type of performance requirement would state that the system should have a CPU utilization rate that does not exceed 50 percent when the system is run with an average workload.

Unfortunately, this type of definition is not applicable to the type of performance testing that this project seeks to perform, as we do not have specific performance requirements. Rather we are seeking to compare performance characteristics of different server architectures under similar workloads in an effort to determine which architecture performs better.

2.4 Performance Monitoring

Performance monitoring is the act of observing a system under test to gain an insight into how the system is performing during testing. One of the key considerations of performance monitoring is a concept known as the “observer effect”. Simply stated this is the potential impact that the act of observing a system may have on the system being observed.

As noted by Helsinger et al [10]:

> Any process for measuring the performance of a system should not itself impact that performance.
>
> The challenge is to balance impact on the system with the expected value of the retrieved data.

Performance monitoring can be broadly categorised into two main types:

- Internal – Monitoring performed from with the server process being tested
- External – Monitoring of system resources (e.g. CPU and memory) performed by external tools

In an effort to ensure that internal performance monitoring will not adversely impact the system under test, the volume of data collected from within the server process will be kept to a minimum. Typically, the type of measurements taken will be timestamps at various key points in the request-response cycle. A global counter of active connections will also be maintained.

2.5 C10K Problem

The C10K problem is succinctly defined by Griffin et al [11] as follows:

> C10k names a limitation of most web servers: they can handle at most 10,000 connections simultaneously.

The most widely cited reference regarding the C10K problem is that of D Kegel [12]. In his article, Kegal proposes a number of potential approaches for dealing with this problem:

- Serve many clients with each thread, and use nonblocking I/O and level-triggered readiness notification.
- Serve many clients with each thread, and use nonblocking I/O and readiness change notification.
- Serve many clients with each server thread, and use asynchronous I/O.
- Serve one client with each server thread, and use blocking I/O.
- Build the server code into the kernel.

It should be noted that all of Kegal’s approaches deal exclusively with thread based server architectures. This is unsurprising since Kegel’s article dates from 2006, when event based server architectures were not as prevalent.

3. System Overview

This section aims to provide a detailed insight into the various components of the systems developed in order to perform the various server push tests. At a high level these fall into four main components:

- Server Instrumentation: The Java and JavaScript code to allow Tomcat and Node.js to accept Long Poll and Comet server push requests and respond accordingly.
- Test Definition: The JMeter test plan which performed the actual load testing on the various servers.
- Test Execution: The automation required to initiate remote monitoring on the system under test, execute the JMeter test plan and collect results from both the system under test and the remote JMeter test servers.
- Report Generation: The parsing and analysis of the raw test data to produce useful summary results and charts.

3.1 Server Instrumentation

Server Instrumentation involved translating the abstract instrumentation plan into server specific code. This instrumentation plan required that an instrumented server provided support for the following calls:

- Time Stamper: A time synchronisation event which allows the load generating system to determine the current time of the system under test. This is required to ensure that time stamped information collected from both the load generator and the system under test can be reconciled.
- Log Manager: Provides the ability to remotely enable/disable server logging. During development it is useful to have log information displayed to the console or written to a log file. However, during actual test execution, this log information is undesirable; the I/O operations required to produce the log output put an extra overhead on the server which may skew test results.
- Event Manger: Provides the ability to create, modify and remove server push event emitters. Server Push technology is based on the idea of a server maintaining many open connections to clients and sending or "pushing" data to each of these clients in response to events on the server. In order to mimic new events occurring, event emitters are required. These emitters periodically generate events which are sent to connected clients.
- Request Manager: Provides support for receiving requests from the load generating system and processing these requests according to the test specification as defined in the POST data of the request. Each request contains information about the request type – Comet or Long Poll, the events that the request is interested in, and a max duration – for Long Poll request. Comet requests do not have a maximum duration, but rather remain open indefinitely.

Figure 1 below illustrates the components developed to support server instrumentation:

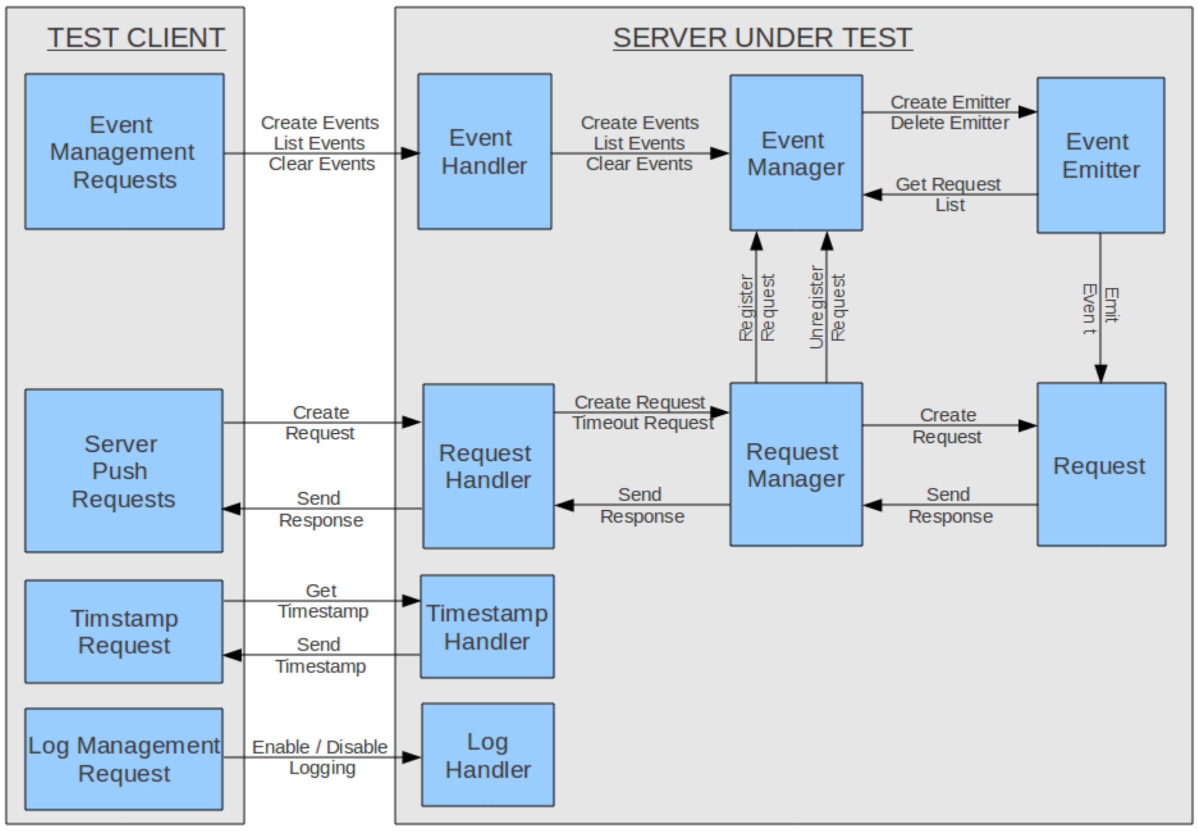


Figure 1: System under test Architecture.

3.1.1 Java Instrumentation for Tomcat

A key goal when developing the Java instrumentation was that as much code as possible was shared between the Tomcat 6 and Tomcat 7 implementations. This was to ensure that any differences in performance could be attributed to the Tomcat specific implementation of Server Push rather than differences in any other part of the instrumentation. This required the development of a shared module which was independent of either Server Push implementation. The majority of the Instrumentation Framework was developed in this shared module with only the actual Server Push request handler requiring specific implementations for each version of Tomcat. To achieve this, the Observer design pattern – implemented using the `java.util.Observer` and `java.util.Obserable` classes – was utilised to communicate event information to the Tomcat specific request handlers.

One of the main challenges here was the management of several thousand threads accessing shared resources to add and remove requests as events were emitted. This required establishing a balance between concurrency control and performance. For example, when an event emitter fires, it needs access to the list of requests which have registered an interest in the event. The emitter must loop over this list and send its event information to each request. At the same time, the request manager is constantly receiving new requests and also requires access to the event emitter's request list in order to add new requests. In the initial implementation the event emitter was synchronising on the request list for the entire emit cycle – which could be several seconds depending on the volume of pending requests. This synchronisation meant that new requests could not be added during the emit loop and so request threads were being blocked until the event loop completed. The solution to this synchronisation issue was to have the event emitter synchronise on the request list for just long enough to make a deep copy of the list and clear the master list. The emitter then used this copy of the list to send event information to each request. This allowed the request manager to continue to add new requests even while an emit cycle was executing.

3.1.1.1 Tomcat 6

Tomcat 6, which was originally released in October 2006 [13], introduced support for Server Push style requests via a number of proprietary classes including `org.apache.catalina.CometProcessor` and `org.apache.catalina.CometEvent`. These classes provide support for an Event Based request model rather than the traditional synchronous request/response model.

This model uses events to indicate when a particular action has occurred on a request. There are four event types which may occur:

- BEGIN: This event is triggered when a request is first received.
- READ: This is where request data is read from the connection. At this point the Request Handler servlet hands the request off to the Request Manager who is responsible for registering the request with all events specified in the request data.
- ERROR: If a request times out – because no event information was received in the allowed time – then the ERROR event is automatically triggered in the Comet Processor. At this point, the Comet Processor will inform the Request Manager that the event has timed out. The Request Manager removes the request from the global request pool and also from any emitters it may be registered with. It then returns timeout response data.
- END: When an event is emitted to a request, the Request Manager sends the response data to the request's response writer. If the request is of type Long Poll, the request is removed from the event emitter and the request's observer is notified which in turn triggers the END event in the Comet Processor. The Comet Processor's END event will inform the Request Manager that the event is ending. The Request Manager removes the request from the global request pool.

For the purposes of the Server Push experiments carried out here, the latest stable version of Tomcat 6 at the time of test execution (v6.0.32 @ Feb 2011) was used [14].

3.1.1.2 Tomcat 7

The Java Servlet 3.0 specification, which was finalised in December 2009, introduced standardised support for Server Push style requests – referred to in the specification as "Async Requests" [15]. The first stable release of Tomcat 7 (7.0.6), which became available in January 2010 [16] was fully compliant with the Servlet 3.0 specification.

This specification introduced support for Async Request by means of the following core classes: `javax.servlet.AsyncContext`, `javax.servlet.AsyncEvent` and `javax.servlet.AsyncListener`.

In order to enable a servlet for Async Request support, the servlet only needs to have the following extra line added to its definition in the project's web.xml: `asyncSupported = true`. Once Async Support has been enabled, a servlet may invoke the `startAsync()` method of a request. This provides access to an `AsyncContext` object which may be used to manipulate the request and response objects. In order to receive notifications of changes to the state of the Async request, the `AsyncListener` interface must be implemented. This interface defined methods for receiving notifications on the following events: `onStartAsync()`, `onComplete()`, `onTimeout()` and `onError()`.

For the Tomcat 7 server instrumentation, a class which implemented the `AsyncListener` interface as well as the `Observer` interface was created. This class, called `AsyncRequestMonitor`, has access to the Request Manager as well as the AsyncContext.

The following methods were implemented in this class:

- onStartAsync: When a new request is received the `onStartAsync()` method of `AsyncRequestMonitor` is used to add the request to the Request Manager.
- onTimeout: If a request times out, the `onTimeout()` method of `AsyncRequestMonitor` is called. This in turn calls the Request Manager. The Request Manager removes the request from the global request pool and also from any emitters it may be registered with. It then returns timeout response data which is sent to the client.
- update: When an event is emitted to a request, the Request Manager sends the response data to the request's response writer. If the request is of type Long Poll, the `update()` method of the `AsyncRequestMonitor's Observer` interface is called.
- onComplete: The update method above calls the `onComplete()` method of the `AsyncListener` interface. The triggering of the `onComplete()` method causes the request to be removed from the Request Manager's global request pool.

For the purposes of the Server Push experiments carried out here, Tomcat version 7.0.11 (May 2011) was initially selected. However, due to a bug [17] with Tomcat's handling of the `setTimeout()` method on `AsyncContext` which was causing Comet requests to end immediately, the version of Tomcat 7 used for testing was upgraded to version 7.0.16 (June 2011) [18].

3.1.2 JavaScript Instrumentation for Node.js

JavaScript is designed from the ground up to be an asynchronous, event driven language. This makes it ideally suited for Server Push style programming. With JavaScript there is no need to look to proprietary vendor solutions (like the CometProcessor in Tomcat 6), or to adhere to published specifications (like Servlet 3.0 in Tomcat 7). Support for asynchronous programming is built into the language.

Node.js provides access to a HTTP module which can be used to create an Asynchronous HTTP server [19]. Request processing is managed through the use of request event handlers such as `data()`, `end()` and `close()`. After a new request is received, the `end()` handler is called. This handler is used to add the request to the global request pool and to register the request's interest in any events specified in the POST data.

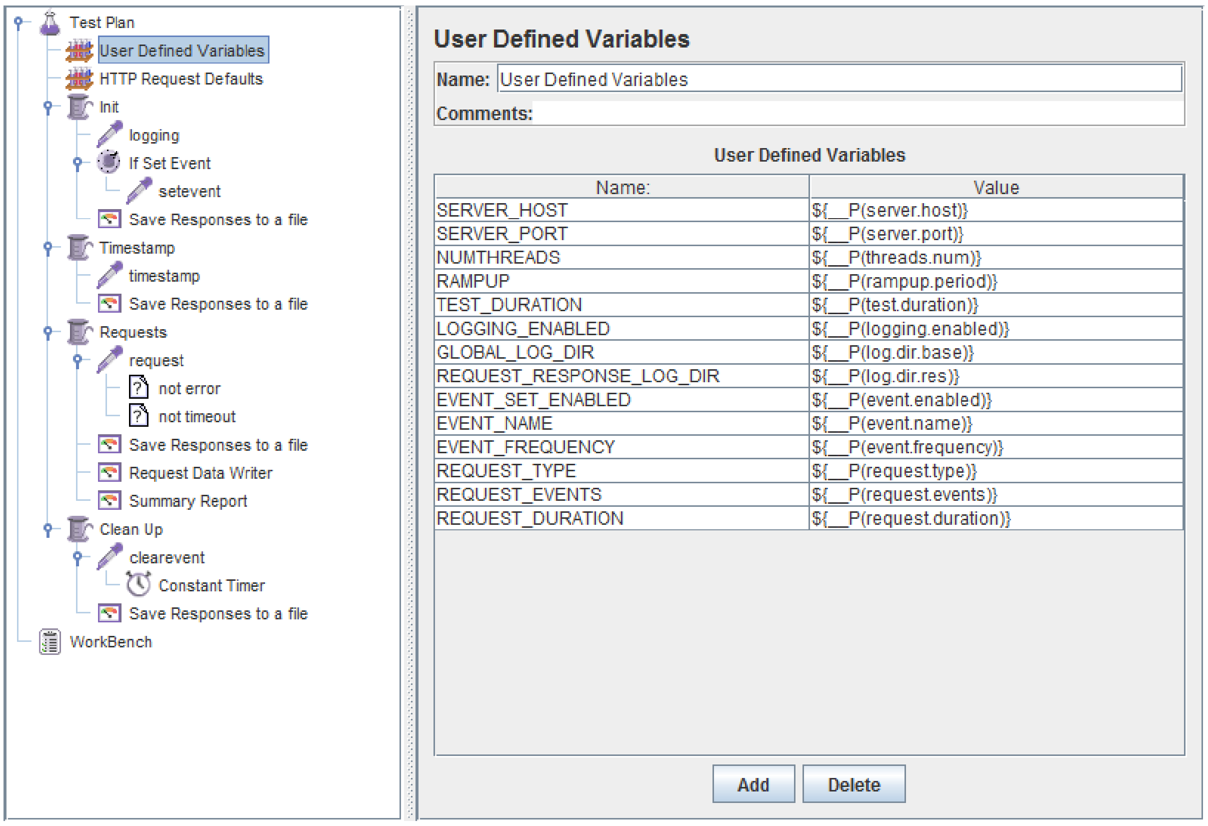
As JavaScript and Node.js are event driven, it is relatively easy to build support for event emitters which fire at regular intervals. When an event emitter is created, the standard `setTimeout()` method in JavaScript is used to allow the event's emitter loop to call itself. When the emit loop fires, any active requests which are registered for the event get data emitted to them. If the request is of type Long Poll, the request is removed from the emitter and the response is committed. If the request is of type Comet, the event data is flushed to the response's writer and the request remains registered for the event.

Providing support for Long Poll request time outs proved to be more complex than anticipated. While Node.js does provide support for setting a timeout on the request, this proved to be unsuitable for the purpose of notifying the Request Manager that a request had timed out. The reason for this is that by the time the request's `timeout()` event had fired, the response object had already been discarded. This meant that it was not possible to send timeout response data. To overcome this, the standard `setTimeout()` method in JavaScript was once again used. A timeout equal to the maximum duration specified in the request data was used to set up a call to a function which would send a timeout response. This function first checked if the request had already been removed (due to an emitter sending data to the request), and if not, the timeout response was sent and the request was removed from the global pool. Similarly, the event emitter checked each request to see if had already timed out (by querying the global request poll), and if so, the request was removed from the emitter without any attempt to send event data.

For the purposes of the Server Push experiments carried out here, the latest stable version of Node.js at the time of test execution (v0.4.10 @ July 2011) was used [20].

3.2 Test Definition

To provide support for the various types of tests which were required (Comet, Long Poll, with events, without events) it was necessary to develop a highly configurable test plan. Apache JMeter version 2.4 was used to define and execute this test plan. The screen shot below shows the complete JMeter test plan which was developed.



- Test Plan
  - User Defined Variables
  - HTTP Request Defaults
  - Init
    - logging
    - If Set Event
      - setevent
    - Save Responses to a file
  - Timestamp
    - timestamp
    - Save Responses to a file
  - Requests
    - request
      - not error
      - not timeout
    - Save Responses to a file
    - Request Data Writer
    - Summary Report
  - Clean Up
    - clearevent
      - Constant Timer
    - Save Responses to a file
- WorkBench

User Defined Variables

Name: User Defined Variables

Comments:

User Defined Variables

Name:	Value
SERVER_HOST	${__P(server.host)}
SERVER_PORT	${__P(server.port)}
NUMTHREADS	${__P(threads.num)}
RAMPUP	${__P(rampup.period)}
TEST_DURATION	${__P(test.duration)}
LOGGING_ENABLED	${__P(logging.enabled)}
GLOBAL_LOG_DIR	${__P(log.dir.base)}
REQUEST_RESPONSE_LOG_DIR	${__P(log.dir.res)}
EVENT_SET_ENABLED	${__P(event.enabled)}
EVENT_NAME	${__P(event.name)}
EVENT_FREQUENCY	${__P(event.frequency)}
REQUEST_TYPE	${__P(request.type)}
REQUEST_EVENTS	${__P(request.events)}
REQUEST_DURATION	${__P(request.duration)}

Figure 2: JMeter Test Plan.

To achieve the level of configurability required, it was necessary to define a total of 14 variables which controlled various aspects of the test plan. The details of these variables are outlined below.

Variable Name	Meaning
SERVER_HOST	The IP address of the system under test
SERVER_PORT	The port of the system under test. Each of the 3 server push implementations was bound to a different port
NUMTHREADS	The maximum number of request threads to use during test execution
RAMPUP	The number of seconds to spend ramping up to the NUMTHREADS value
TEST_DURATION	The total number of seconds to execute the test for
LOGGING_ENABLED	A Boolean indicator as to whether logging should be enabled on the

	system under test
GLOBAL_LOG_DIR	The log directory where JMeter records summary results
REQUEST_RESPONSE_LOG_DIR	The log directory where JMeter saves individual responses
EVENT_SET_ENABLED	A Boolean indicator as to whether a server push event should be initialised
EVENT_NAME	The name of the server push event to create (only used if EVENT_SET_ENABLED = true)
EMIT_INTERVAL	The interval in milliseconds between emit cycles for the server push event "EVENT_NAME" (only used if EVENT_SET_ENABLED = true)
REQUEST_TYPE	The request type to send to the system under test – Long Poll or Comet
REQUEST_EVENTS	The server push events that the request should subscribe to
REQUEST_DURATION	The maximum duration of the request (only used if REQUEST_TYPE = Long Poll)

The test plan utilises several JMeter elements to control the test execution. The structure of the test plan is described below:

User Defined Variables: As described above, these are used to configure and control test execution.

HTTP Request Defaults: This element is used to configure common defaults for all HTTP requests such as the server IP address and server port.

"Initialisation" Thread Group: Thread groups are used to control the number of executions of each element contained within the thread group. In the case of the initialisation thread group, each element is configured to execute exactly once.

"Logging" HTTP Request: This request uses the LOGGIN_ENABLED variable to enable/disable logging on the system under test.

"If Set Element" Logic Controller: This conditional element checks the EVENT_SET_ENABLED variable to determine if an event emitter request should be sent to the server.

"Set Event" HTTP Request: This element uses the EVENT_NAME and EVENT_FREQUENCY variables to initialise an event emitter on the system under test – but only if the "If Set Element Logic Controller" evaluated to true.

Save Responses to a file: This element, which occurs within each thread group, is used to save the HTTP responses from the system under test to files on disk. This element uses the REQUEST_RESPONSE_LOG_DIR variable to determine where to save response files to.

"Timestamp" Thread Group: This thread group, which contains the Timestamp HTTP Request, is also configured to execute exactly once.

"Timestamp" HTTP Request: This element is used to capture the current timestamp on both the client and server. The request sends the current timestamp from the test client to the server – using the JMeter function `${__time}` to initialise the current time on the client. The server responds with its current timestamp. This information is used during report generation to synchronise time stamped data collected from both the client and server.

"Request" Thread Group: This thread group is at the heart of the test plan. It uses the following variables for configuration purposes:

- NUMTHREADS to configure how many concurrent requests the JMeter client should make to the server.
- RAMPUP to configure how many seconds the client should take to ramp up from 0 to NUMTHREADS.
- TEST_DURATION to configure how long the main body of the test should execute for.

"Request" HTTP Request: This element is responsible for performing the actual Server Push request against the server. It uses the REQUEST_TYPE, REQUEST_EVENTS and REQUEST_DURATION variables to configure the POST data to send to the server.

"Not Error" Response Assertion: This element checks the HTTP response from the system under test to ensure that it does not contain the string "error". If a JMeter client fails to establish a connection to the system under test – due to excessive load or other connectivity issues – the Java stack track from JMeter which will be saved as the response will always contain the string "error".

"Not Timeout" Response Assertion: This element checks the HTTP response from the system under test to ensure that it does not contain the string "timeout". However, this Response Assertion is complicated by the fact that "timeout" is a valid response if an event emitter was deliberately not configured. To support this, the Response Assertion uses a scripting language called BeanShell [21] to first check whether an event emitter was configured on the server. Only if this check evaluates to true is the HTTP response checked to see if it contains the string "timeout".

Request Data Writer: This element writes summary response data such as request duration, number of active threads, success/failure indicator and the path to the HTTP response data.

Summary Report: This element is only used if JMeter is run in GUI mode (used when initially defining the test plan). It provides a quick overview of the status of a running test – very similar to the information generated by the Request Data Writer.

"Clean Up" Thread Group: This thread group, which executes only once, is used to stop any event emitters which are active on the system under test.

"Clear Event" HTTP Request: This element calls the /event/clear end point on the system under test to stop all emitters.

Constant Timer: This timer is used to delay the "Clear Event" HTTP Request for a period of time equal to the value of the EVENT_FREQUENCY variable. This is used to ensure that the "Clear Event" request is not fired while there are still responses pending on the server.

3.3 Test Execution

3.3.1 Overview

The main components required to execute a test run were identified as follows:

Server Push Test Runner: This component was responsible for orchestrating the entire test run. Its tasks included setting up required log directories, initiating monitoring on the system under test, launching JMeter and collecting test results.

JMeter Core: In order to achieve the required volume of concurrent requests it was necessary to run JMeter in what is called "distributed mode" [22]. In this testing mode, there is a single core JMeter process which controls several distributed JMeter Servers.

JMeter Servers: These servers were controlled by the JMeter core process and were responsible for performing the actual load test against the system under test.

System Under Test: The server which was running the Server Push software.

Due to the fact that several hundred test runs were required to gather all the test data, it was decided quite early that the test execution process should be automated as much as possible. To achieve the required automation, a shell script was created to represent the **"Server Push Test Runner"** component. This script controlled the setup, execution and tear down of a test run. The following diagram outlines the flow of a typical test run:

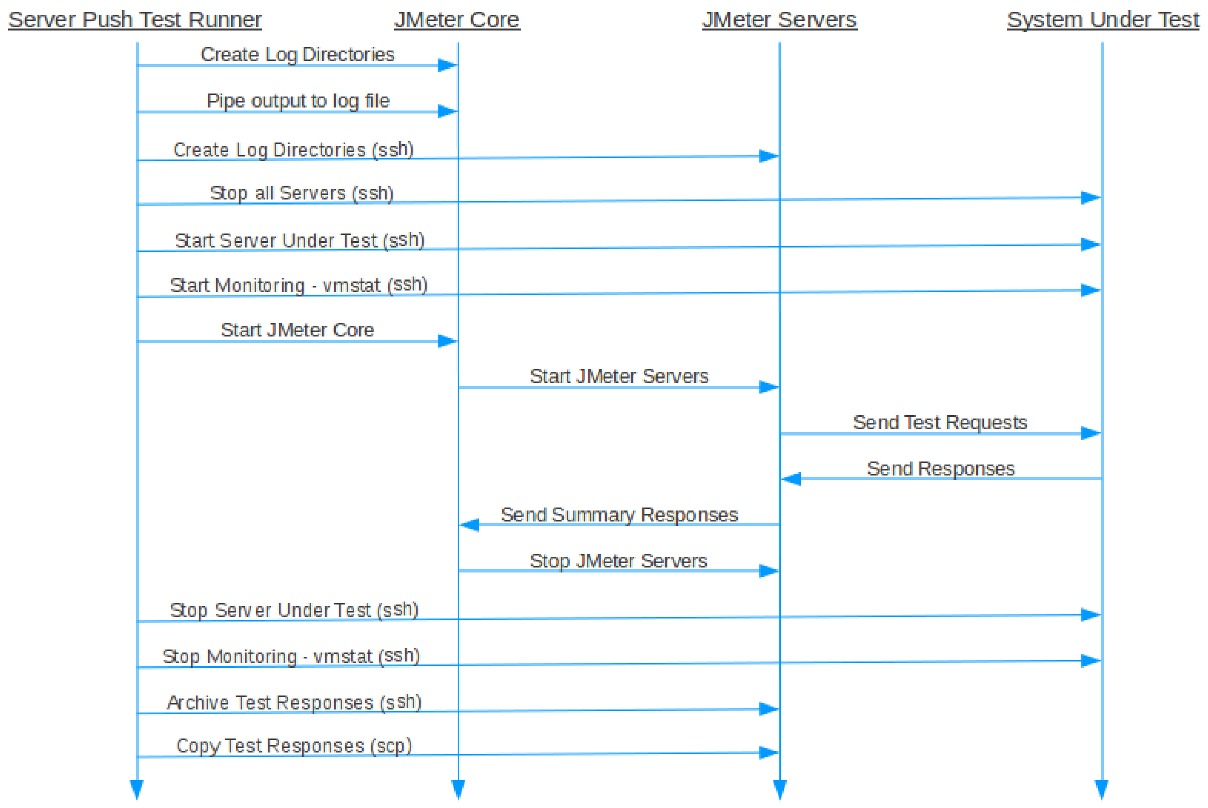


Figure 3: Test Runner Script.

3.3.2 Test Runner Script

To execute a test run, the Server Push Test Runner script (called run.sh) can be called as follows:

```
./run.sh <server-type> <rampup> <duration> <threads> <test-type> [<max-duration>
<event-name> <emit interval>]
```

The parameters to the Server Push Test Runner script have the following meanings:

- **<server-type>** : The type of the server being tested:
  - node = Node.js
  - tc6 = Tomcat6
  - tc7 = Tomcat7
- **<rampup>** : The time in SECONDS during which all threads for the test will be created.
- **<duration>** : The duration in SECONDS to run the test for.
- **<threads>** : The NUMBER of threads to use.
- **<test-type>** : The type of test to perform:
  - comet
  - poll
- **<max-duration>** : The duration in MILLISECONDS of each poll test case (Required if the test type is poll).
- **<event-name>** : The name of the event to create and use for the requests (Optional – If omitted, no event emitter will be created).
- **<emit-interval>** : The frequency with which the events are emitted (Optional – If omitted, no event emitter will be created).

For example:

- **./run.sh tc6 60 900 5000 comet** – A 15 minute Comet test, with a 1 minute ramp up, against Tomcat 6, using 5000 threads.
- **./run.sh tc7 30 600 5000 poll 3000, event1, 2000** – A 10 minute Long Poll test, with a 30 second ramp up, against Tomcat 7 with a Long Poll duration of 3 seconds, and an event emitter which emits every 2 seconds.
- **./run.sh node 90 1200 5000 poll 3000** – A 20 minute Long Poll test, with a 90 second ramp up, against Node.js with a Long Poll duration of 3 seconds. In this case no event emitter is specified so all requests will time out after 3 seconds.

3.3.3 Test Execution Steps

The steps required by the Server Push Test Runner to execute a complete test run (as outlined in the diagram on the previous page) are explained in detail below:

Create Log Directories: In order to ensure that test results for each test were fully isolated, the first step in a test run was to create a directory for JMeter Core to write log files to. For ease of identification, the naming convention for this log directory took following form:

/<base-dir>/<date>-<server-type>-<request-type>-<num-threads>-<duration>

This directory name became the value for the variable GLOBAL_LOG_DIR which was passed to JMeter.

Create Log Directories (ssh): Due to the fact that the JMeter Servers were not all located on the same physical machine as the Server Push Test Runner, and the fact that JMeter will not create log files if the specified log directory does not exist, it was necessary to establish secure shell (ssh) [23] connections to each JMeter Server in order to create the required log directories.

Stop All Servers (ssh): To ensure that the System Under Test was operating at maximum capacity, all three of the servers (Tomcat 6, Tomcat 7 and Node.js) were issued with stop commands before a test began.

Start Server Under Test (ssh): The specific server which was being tested as part of the current test run was started.

Start Monitoring (ssh): To facilitate remote monitoring, the vmstat [24] application was run against the system under test over an ssh connection. The command used to monitor remote system usage is as follows:

```
vmstat -SM -a 1
```

The options passed to vmstat were used to perform the following configuration:

- -SM: Format the memory usage to display Megabytes rather than Kilobytes.
- -a: Display active/inactive memory rather than buffer and cache memory stats.
- 1: Repeat every 1 second until interrupted.

Start JMeter Core: JMeter Core was launched in a distributed mode with all required parameters passed on the command line. This included information on the IP addresses of the various JMeter servers as well as the 14 parameters required by the JMeter Test Plan.

Wait: The test runner script waited until the JMeter test had completed. During the wait period the summary response file generated by JMeter was polled every 5 seconds for new information. The test runner script continues to wait until this file was no longer being updated. To ensure that updates to the file had completed, the script rechecked the file a second time after it initially determined that the updates to the file had ceased.

Stop JMeter Core: Once the test runner script had finished waiting, it issued a command to kill the JMeter core engine. This was a defensive measure to ensure that JMeter was not continuing to run in the background and consume system resources.

Stop Server Under Test (ssh): After the test run had completed, the server under test was stopped to reduce resource usage on the hardware hosting the system under test.

Stop Monitoring: After launching JMeter Core, the Server Push Test Runner script would sleep for the duration of the test execution. Once that time period had elapsed, the vmstat monitor running against the system under test was stopped.

Archive Test Responses: Depending on the duration of the test run, and the number of active threads, there were often 10's or even 100's of thousands of test responses per JMeter server. Despite the fact that all hardware was running on the same local network, copying each file individually was incredibly inefficient. To improve this performance, the Server Push Test Runner script was enhanced to make an ssh connection to each JMeter server in order to create a single tar file archive [25] of all test responses.

Copy Test Responses: Once all test responses had been archived, the scp [26] application was used to copy the response archives from each JMeter Server back to the Server Push Test Runner. This ensured that all data relating to a test run was stored centrally within a single directory.

3.3.4 Test Framework Hardware

The following hardware was used to facilitate the test execution:

1.

CPU: Intel Core 2 Duo @2.53 GHz

Memory: 4.00 GB RAM

Operating System: Ubuntu 10.04

Used For: Server Push Test Runner, JMeter Core, 1 x JMeter Server

2.

CPU: Intel Pentium 4 @ 3GHz

Memory: 4.00 GB RAM

Operating System: Ubuntu 10.04

Used For: JMeter Server

3.

CPU: Intel Celeron Dual Core @ 2.1GHz

Memory: 4.00 GB RAM

Operating System: Ubuntu 10.04

Used For: JMeter Server

4.

CPU: Intel Core 2 Duo @ 2.53GHz

Memory: 4.00 GB RAM

Operating System: Ubuntu 10.04

Used For: JMeter Server

5.

CPU: Intel Core2 Quad @ 2.33GHz

Memory: 4.00 GB RAM

Operating System: Ubuntu 10.10

Used for: System Under Test

3.3.5 Operating System Selection

During the initial phases of testing, two of the JMeter Servers were running versions of the Windows Operating Systems. This Operating System does not natively provide support for ssh. To overcome this it was necessary to first install Cygwin on the Windows based hardware. Cygwin is “a collection of tools which provide a Linux look and feel environment for Window” [27]. Once Cygwin was installed, it was also necessary to install and configure Open SSH [28] as well as configuring public/private ssh keys [29] on the various boxes to allow the Server Push Test Runner to connect to the other components without requiring username and password authentication.

During the testing phase, it quickly became clear that the Windows Operating System was not suitable for the high volumes of concurrent requests being made against the system under test. This manifested itself through frequent Out Of Memory errors in JMeter. Despite best efforts to tune the memory allocated to JMeter, it was not possible to achieve a stable test framework using Windows. To overcome this, Ubuntu 10.04 was installed as a dual boot operating systems on all hardware running the Windows Operating System. Ubuntu 10.04 was chosen over the more recent versions of the Ubuntu Operating System as it matched the Operating System version on the test runner.

3.4 Report Generation

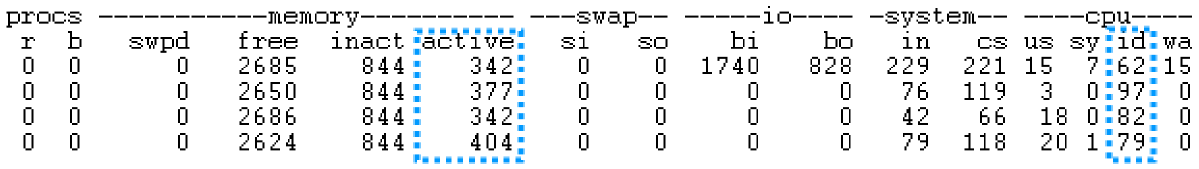
At the end of each test run, a suite of reports were automatically generated to provide a summary analysis of the data gathered during the test run. In order to generate these reports, information from various sources (vmstat logs, summary response data, event emitter logs, HTTP responses etc) was analysed. Due to the large volume of individual response files generated during a test run, it was decided to use Node.js to perform the data analysis.

3.4.1 Data Analysis

Data analysis required parsing the test results and generating JSON formatted data structures which could be used by the charting tool selected – Open Flash Chart [30]. In order to provide support for generating comparison charts between different test runs, e.g. comparing CPU or memory usage at varying request levels and/or on various server types (Tomcat 6, Tomcat 7 and Node.js), the generated JSON data set was also saved independently from the final report data. This allowed for data sets from different test runs to be drawn together to produce final comparison charts.

3.4.1.1 vmstat

The information captured by the vmstat application is fixed width, space delimited rows of data. The following screen shot shows an example of vmstat data. The columns highlighted in blue represent the amount of active memory in megabytes, and the percentage of CPU which is currently idle.



```
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free  inact active   si   so    bi    bo   in   cs us sy id wa
 0  0      0   2685    844    342    0    0  1740   828  229  221 15  7 62 15
 0  0      0   2650    844    377    0    0     0     0   76  119  3  0 97  0
 0  0      0   2686    844    342    0    0     0     0   42   66 18  0 82  0
 0  0      0   2624    844    404    0    0     0     0   79  118 20  1 79  0
```

In order to extract the required information from the vmstat log files, the fixed width, space delimited, data was run through the following regular expressions:

- `dataLine.replace(/\s{2,}/g, ' ');` – Replaces all occurrences of two or more spaces with a single space.
- `dataLine.replace(/^\s+|\s+$/g, '');` – Trims any leading and trailing whitespaces from the row of data.

After applying these regular expressions, the data was then in a format where it could be split into an array of elements; using a single white space as the delimiter: `dataLine.split(/\s/g);`

In order to represent the captured information on a chart as percentage values the following calculations were performed:

- % Memory used = (<Active Memory> / <Total Memory:4096MB>) * 100
- % CPU Used = 100 - <Idle CPU>

Due to the fact that vmstat measures overall system resource usage, "Active Memory" refers to total active memory (including memory allocated to the Operating System, and background processes), not specifically memory used by the server under test. While every effort was made to keep background processes to a minimum, some variation in background memory usage was unavoidable. Background memory usage was typically between 10% and 15% during tests.

3.4.1.2 HTTP Response Data

The response data sent by the system under test included all information required to analyse server performance from the inside. An example HTTP Response from a system under test is shown below:

```
{
   response: {
      event: {
         timings: {
            duration: <Time (ms) since event emit started until event sent to this request>,
            sent: <Timestamp of when the event was emitted to this request>,
            created: <Timestamp of when the event emit loop started>
         },
         name: <Name of the event currently emitting>,
         connections: {
            total: <Total number of connections being emitted to>,
            place: <Position in the queue for this request>
         }
      }
   },
   request: { <Request data received> },
   timings: {
      duration: <Time (ms) since request was received until event sent to this request>,
      sent: <Timestamp of when the request manager sent the response to this request>,
      created: <Timestamp of when the request was first received by the request manager>
   },
   connections: {
      finish: <Number of connections across all events when the response was sent>,
      start: <Number of connections across all events when the request was first received>
   }
}
```

For the purpose of the test results required, four key pieces of information from the response data were extracted. These were:

1. **timings.created:** The timestamp at which the request object was created on the server. This time stamp was required so that the `connections.start` value mentioned in point 2 below could be correlated with the vmstat CPU and memory usage information. This allowed the number of concurrent connections versus resource usage to be graphed.

2. **connections.start:** The number of active requests when this request was received. The reason that the connection information was taken from the individual response files rather than the JMeter summary results is that each JMeter server only reports active connections for itself. This means that if 3 JMeter Servers were each making 2,000 concurrent connections, the maximum value for concurrent connections which would appear in any JMeter log file is 2,000. By comparison, the system under test would report the true value for active connections – 6,000 in this example.

3. **event.timings.duration:** The number of milliseconds which elapsed between the time that an event emitter fired and the event data was sent to this request.

4. **event.connections.place:** The position in the event emitters list that this request occupied. A graph of `event.timings.duration` versus `event.connections.place` was used to compare the speed and performance of event emitters across the different Server Push implementations.

3.4.2 Chart Definition

As mentioned previously, the chart generation tool selected was Open Flash Charts 2. This tool accepts a JSON format data file as input and uses an Adobe Flash .swf file to render the input data as a chart.

For the purpose of the tests carried out, line charts were used exclusively to render results. This format ideally suited the test data as it allowed various values (CPU usage, memory usage, active threads, event emit information etc) to be graphed over time.

The basic structure of the Open Flash Charts data file is outlined below:

```
{
  elements: [
    {
      type: "line",
      colour: "#DB1750",
      values: [
       < Output from data analysis injected here >
      ],
      dot-style: {
        type: "dot",
        colour: "#f00000"
      },
      width: 1
    }
  ],
  title: {
    text: "Chart Title"
  },
  x_axis: {
    min: 0,
    max: 100,
    steps: 10,
    labels: {
      steps: 10,
      visible-steps: 2
    }
  },
  y_axis: {
    min: 0,
    max: 100,
    steps: 10,
    labels: {
      steps: 10,
      visible-steps: 2
    }
  },
  bg_colour: #FFFFFF
}
```

As part of report generation for each test run, the default values for the following items were updated to reflect the summary data generated during data analysis:

- x_axis.max
- y_axis.max
- title.text

4. Test Preparation and Permutations

This section describes the steps taken to ensure that the systems under test were configured for optimal performance prior to testing, with a particular emphasis on the various parameters which were used to configure the Java Virtual Machine. It also describes the various permutations of tests which were carried out and details how these were combined to identify the final test suite.

4.1 Test Preparation

4.1.1 Java Memory Management

The Java Virtual Machine (JVM) is the runtime environment within which Java Applications execute. Tomcat servers are Java based applications, and so can (and should) be configured to make optimal use of the resources (particularly memory) available to them.

There are several parameters which may be passed to the JVM to configure memory usage. A comprehensive list of these parameters is available from the java command line executable. During the course of preparing for test execution, several of these parameters were investigated and tuned.

One of the most commonly used JVM parameters is **–Xmx** which is used to set the maximum Java heap size. The Java *"heap"* is defined as the memory "that is shared among all Java virtual machine threads. The heap is the runtime data area from which memory for all class instances and arrays is allocated."[31].

To the casual observer, it may appear that setting the heap value as large as possible is the best course of action to ensure that the JVM can support the largest possible number of threads/objects etc.

This is not the case however, particularly when it comes to dealing with large numbers of threads. Each new thread that gets created needs to have memory allocated within the JVM for the actual thread itself. This memory is not allocated from the heap, but rather it gets allocated from the *"stack".* The stack is defined as follows: "A Java virtual machine stack is analogous to the stack of a conventional language such as C: it holds local variables and partial results, and plays a part in method invocation and return."[31]. The amount of memory allocated within the stack for each new thread can be controlled using the **–Xss** parameter.

Each thread created within a JVM also has a memory overhead outside of the JVM. This is because Java threads are mapped to native Operating System threads to allow the Operating System to manage scheduling of threads.

To achieve optimal performance, a balance must be struck between the amount of memory allocated to the heap (using –Xmx), the amount of memory allocated to each new thread within the JVM (using –Xss) and the amount of memory left available to the Operating System for native thread creation.

The diagram in Figure 4 provides a rough graphical illustration of the balance which is required to achieve optimal performance (note that relative allocations are not to scale):

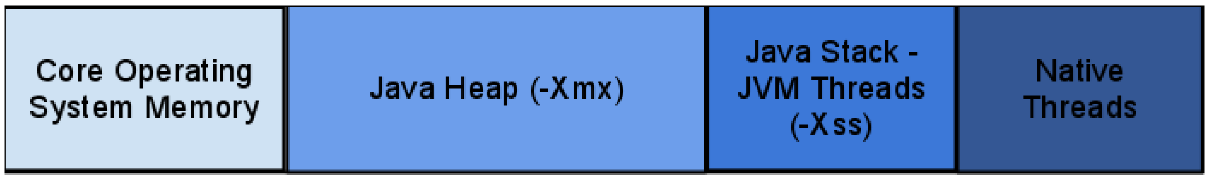


Figure 4: Java Memory Allocation.

To facilitate with basic testing of memory allocation, a simple test application was developed which would recursively call a function to create new threads. This application was implemented as a simple java servlet which could be called via a HTTP request. Once invoked, the servlet recursively called itself, outputting the number of calls made to the command line.

As the goal here was the creation of large numbers of threads, synchronous HTTP requests were used (i.e. non-blocking, asynchronous requests were deliberately not used, as these would have resulted in fewer threads being created).

Depending on the values used for the –Xmx and –Xss parameters, two different Out Of Memory errors were possible:

- java.lang.OutOfMemoryError: Java heap space – This indicated that the value for –Xmx was too small
- java.lang.OutOfMemoryError: unable to create new native thread – This indicated that there was insufficient memory available outside of the JVM to store new threads.

By repeatedly running the basic servlet test outlined above with varying values for –Xmx and –Xss, the following optimal values were identified:

- -Xmx : 2,400m
- -Xss: 64k

These are the values which were then used to configure both tomcat servers for test execution.

4.2 Test Permutations

Tests were executed against three different server types:

- Node.js
- Tomcat 6
- Tomcat 7

Two types of Server Push tests were performed:

- Long Poll – requests are terminated as soon as an event is emitted or after the specified timeout
- Comet – requests are never deliberately terminated and multiple events may be emitted to each request

For each test performed, it was possible to specify whether events should be emitted and if so, with what frequency these emits should occur. The emit interval was varied as follows:

- None – i.e. no events
- 1 second
- 5 seconds
- 10 seconds
- 30 seconds

For Long Poll style tests where no event was emitted, the maximum request duration was varied as follows:

- 1 second
- 5 seconds
- 10 seconds
- 30 seconds

For each test type, the value for concurrent requests was varied as follows:

- 1,000
- 5,000
- 10,000
- 15,000
- 20,000

In total, there were 195 unique tests carried out:

For both Comet and Long Poll style requests the following tests were performed:

- 3 Server Types (Node.js, Tomcat 6, Tomcat 7)
- 2 Server Push styles (Comet, Long Poll)
- 4 variations for emit interval (1 second, 5 seconds, 10 seconds, 30 seconds)
- 5 variations for concurrent connections (1000, 5000, 10000, 15000, 20000)

This accounted for 120 tests.

For Comet requests with no events emitted, tests were performed with various values for concurrent connections:

- 3 Server Types (Node.js, Tomcat 6, Tomcat 7)
- 1 Server Push style (Comet)
- 1 variation for emit interval (None)
- 5 variations for concurrent connections (1000, 5000, 10000, 15000, 20000)

This accounted for 15 tests.

For Long Poll requests with no events emitted, tests were performed with various values for maximum request duration:

- 3 Server Types (Node.js, Tomcat 6, Tomcat 7)
- 1 Server Push style (Long Poll)
- 4 variations for request duration (1 second, 5 seconds, 10 seconds, 30 seconds)
- 5 variations for concurrent connections (1000, 5000, 10000, 15000, 20000)

This accounted for 60 tests.

The results in section 5 relate to a single test run. Ideally all tests would have been run multiple times and charts would have been generated based on average results. Unfortunately, time constraints did not allow for this. However, in any situation where anomalous results were recorded, the tests were re-run at least once to confirm the results.

Appendix A provides a full list of all 195 test combinations executed.

5. Results and Findings

This section explores the tests executed and the results gathered. These results are presented from various perspectives. Due to the volume of unique tests executed not all test results are displayed in this section. To facilitate comparative analysis, results from individual test runs have been amalgamated into summary charts.

The results in this section are divided into two main categories.

In the first category results are presented in the form of a comparison between the two Server Push styles tested. These comparisons are divided into the following sub-categories:

- Comet – No Events
- Comet – With events
- Long Poll – No Events
- Long Poll – With Events

For each of these categories, performance and resource usage was been analysed for various numbers of concurrent connections.

For tests where event emitters were active, the effect of varying the emit interval was also analysed.

For Long Poll style requests with no events, the effect of varying the Log Poll request duration was also analysed.

In the second category each of the three server implementations (Node.js, Tomcat 6 and Tomcat 7) were analysed to determine their optimal performance.

All tests were run for 600 seconds (i.e. 10 minutes). In some cases, the test took some additional time to complete. The maximum time reported in any chart has been capped at 700 seconds. If the test completed in less than 700 seconds, then the chart will not display the full 700 seconds, but rather will display information up to the test duration. For each chart, the labels on the x axis represent elapsed seconds.

For charts that show memory usage, it should be noted that the value displayed is total memory used on the server hosting the system under test. Due to other processes running on the system, and natural memory usage variations, the initial value for memory usage may vary somewhat (typically starting values for memory usage are in the range of 10% to 15%).

In order to ensure a fair and accurate comparison of memory usage in the charts below, any variations in the initial memory usage of the components being compared has been removed.

To facilitate visual comparisons between different test runs, multiple charts have been shrunk and combined in the sections below. Appendix B provides full size versions of each chart used in the following sections.

5.1 Server Push Comparisons

This section reviews each of the Server Push implementations – Comet and Long Poll. For each chart in this section, the performance of the three server implementations (Node.js, Tomcat 6 and Tomcat 7) are graphed against each other.

5.1.1 Comet – No Events

In this test category, Comet style requests were performed against the various servers. No event emitters were configured for these tests which resulted in the majority of Comet requests remaining active on the server for the entire test duration. As was expected, some requests were killed due to errors.

The charts below provide an analysis of Memory usage for these tests:

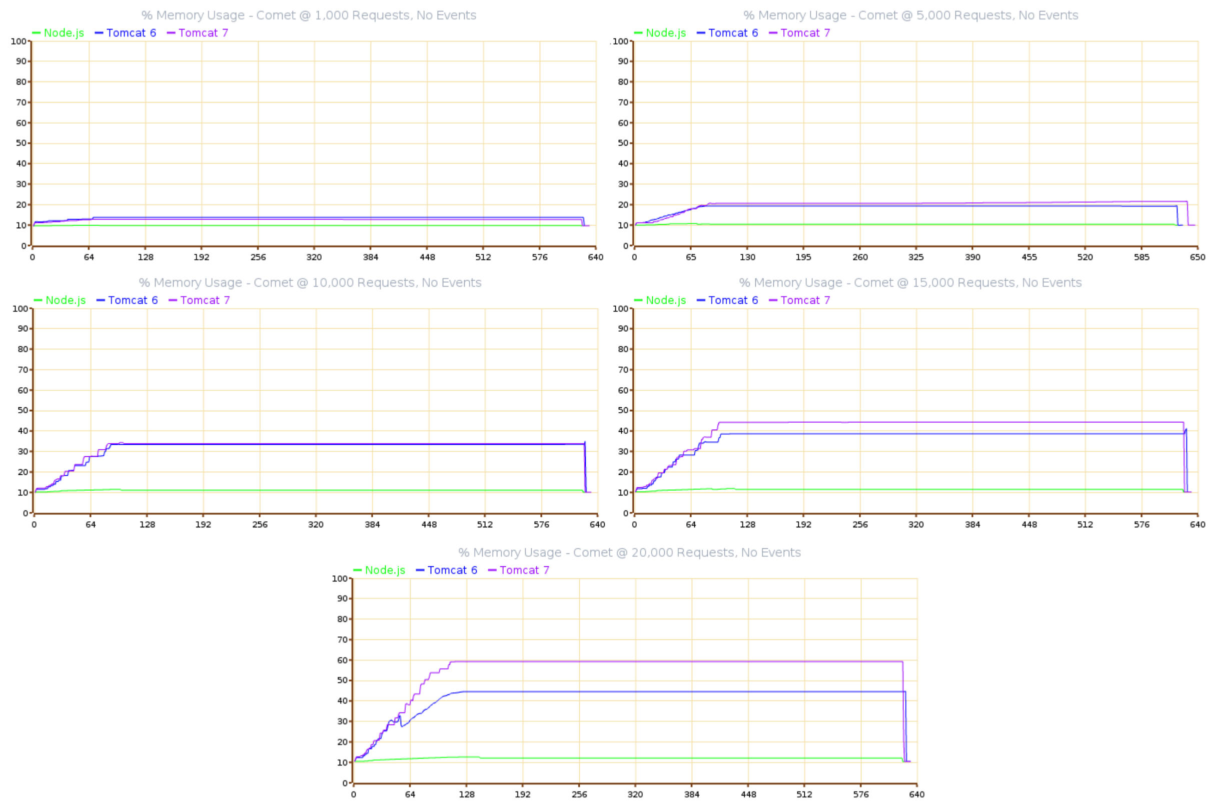


Figure 5: Comet – No Events – Memory Usage.

As the above data shows, there was very little variance in memory usage for Node.js regardless of the number of concurrent connections.

However, Tomcat 6 and 7 displayed a significant increase in memory usage as the number of concurrent connections increased. It is worth noting that the memory increase exhibited by the two Tomcat servers is not linear – there is an initial steep climb followed by a levelling off in memory usage. This is accompanied by increased CPU usage during the steep memory climb (Figure 6). An analysis of java thread dumps performed during testing revealed that the number of HTTP request processing threads (which receive new client requests and asynchronously hand them off for processing) totalled approximately 400 – regardless of the number of concurrent requests. These threads would be early in the test as the number of concurrent requests grew. This accounts for the initial steep climb in memory usage – as each new thread has a memory overhead, as well as the initial CPU spikes which would be associated with the processing overhead of creating these threads.

It is interesting to note that the memory used by Tomcat 7 at 20,000 concurrent connections was approximately 10% higher than Tomcat 6. This is somewhat surprising as one would expect that memory management in Tomcat 7 would have been improved over previous versions.

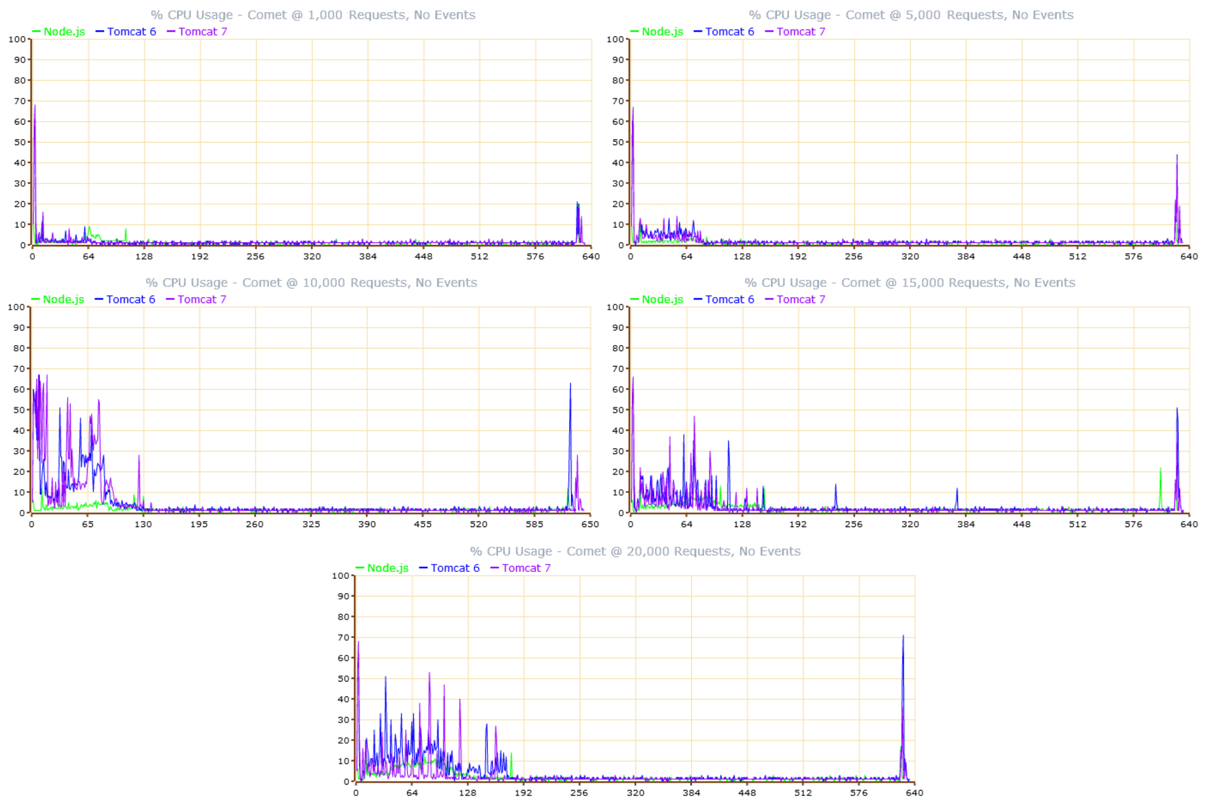


Figure 6: Comet – No Events – CPU Usage.

As can be seen from the above charts, as the number of concurrent connections increased, there was an increase in CPU activity during the early part of the test. This CPU activity is significantly more pronounced for Tomcat 6 and 7 than for Node.js.

The most plausible explanation for the increased CPU usage during the early part of these tests is that there is a CPU overhead associated with new requests being received. For Tomcat 6 and 7, this increase is more pronounced; probably due to the fact that these servers have the additional overhead of thread creation.

Once all requests were established, CPU usage returned to almost zero for all servers. This is expected as there is no CPU overhead associated with maintaining open connections when no data is being sent to connected clients.

5.1.2 Comet – With Events

In this test category, Comet style requests were performed against the various servers. A single event emitter was configured for these tests.

The first set of charts below shows a comparison of CPU usage for 5,000 concurrent requests. In this scenario the frequency at which events were emitted was varied between 1 and 30 seconds.

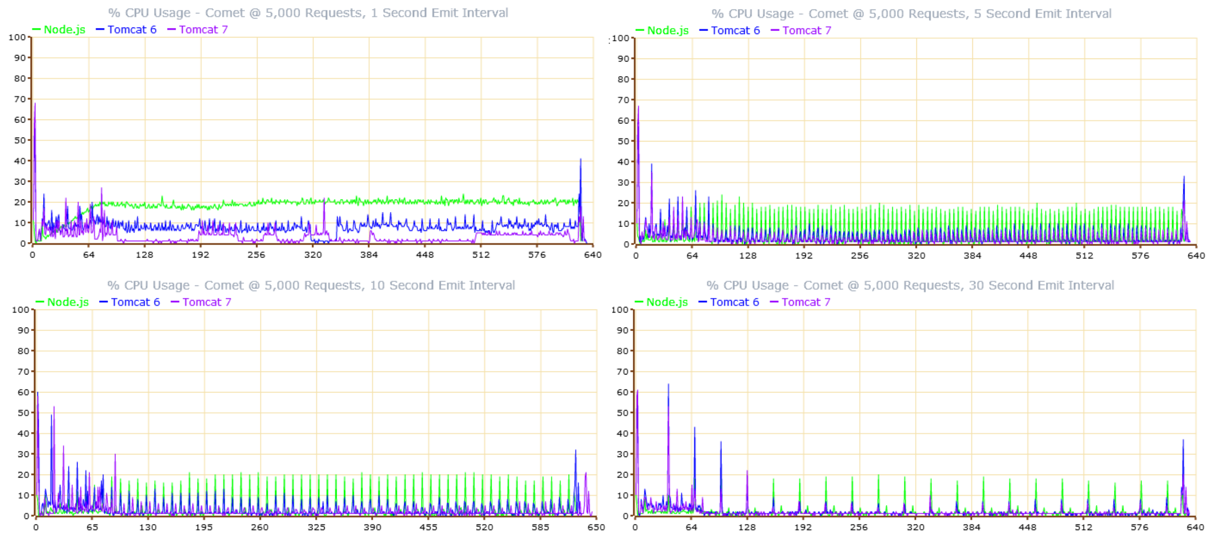


Figure 7: Comet – With Events – 10 Second Emit Interval – CPU Usage.

As can be seen from the above charts, as the event emit interval decreased, the CPU usage increased. This is to be expected as the event emit process (which is iterating over all connections) is a CPU intensive activity.

It is worth noting that the CPU usage for Tomcat 7 in particular dropped to almost zero several times in the first chart, i.e. when the emit interval was 1 second. This can be explained by studying the emit duration, i.e. the length of time taken to complete an emit loop.

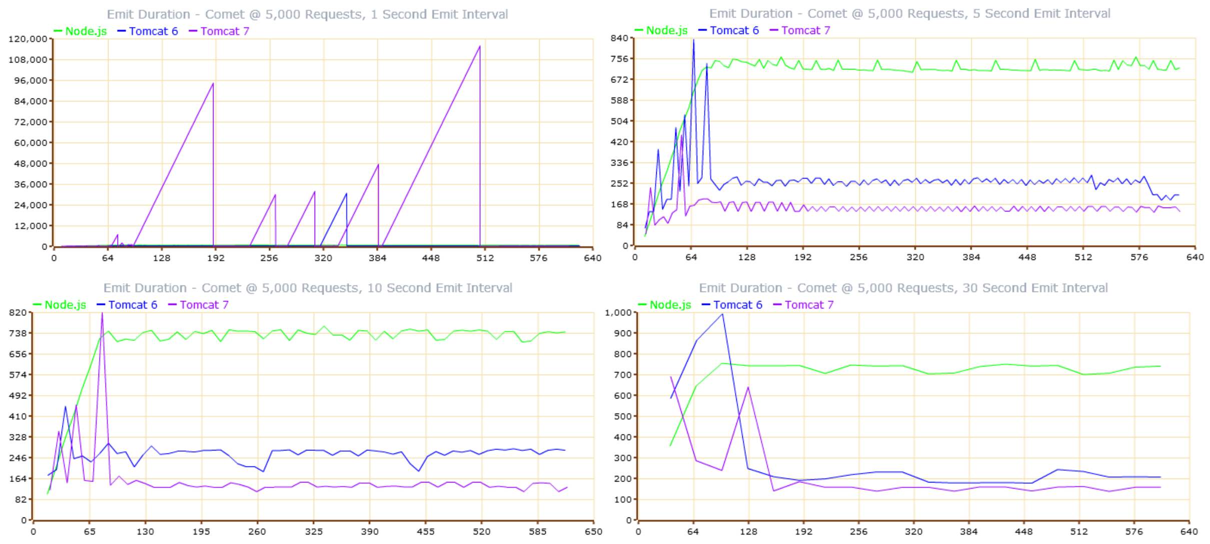


Figure 8: Comet – With Events – 5,000 Requests – Varying Emit Duration.

As can be seen from the Emit Duration data, when the emit interval was 1 second, Tomcat 7 displayed some very erratic behaviour; in several cases the length of time taken to complete an emit loop spiked to approximately 30 seconds, while in the worst case, this time jumped to a massive 115 seconds. These spikes in emit interval coincided with a drop in CPU usage towards 0%. This pattern was also observed periodically in Tomcat 6.

The most probable explanation for this behaviour is that the Java Virtual Machine garbage collector was running at these times. During an emit loop there are several short lived objects created per request which would quickly consume large amounts of memory. As previous charts have demonstrated, there was already a significant memory overhead associated with each request object which would have reduced the amount of available memory, thereby forcing the garbage collector to run more frequently.

As the emit interval was increased, a more stable pattern emerged which showed Node.js consistently taking around 4 times longer to complete an emit loop. This can be explained by the fact that some of the work associated with sending event information to connected clients was handled by other threads in Tomcat 6 and 7. By comparison, the single threaded nature of Node.js meant that all work associated with sending event information to connected clients had to happen within the context of the emit loop.

As the findings below show, the emit duration increased across all servers as the number of connected clients increased. These charts also demonstrate that the pattern identified in Figure 8, which showed Node.js taking around 4 times longer than Tomcat 6 or 7 to complete an emit loop, remained consistent regardless of the number of connected clients.

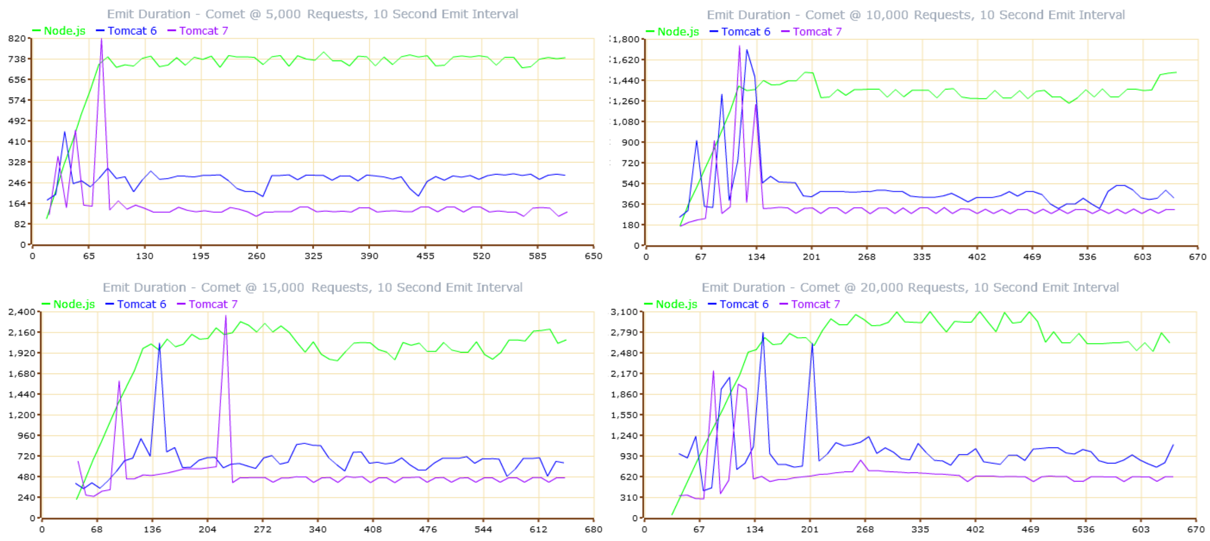


Figure 9: Comet – With Events – 10 Second Emit Interval – Varying Requests.

5.1.3 Long Poll – No Events

In this test category, Long Poll style requests were performed against the various servers. No event emitters were configured for these tests which resulted in each Long Poll request timing out after the specified interval.

The charts below show the CPU usage associated with Long Poll requests at various numbers of concurrent requests. In these charts, the Long Poll requests had a 10 second maximum duration.

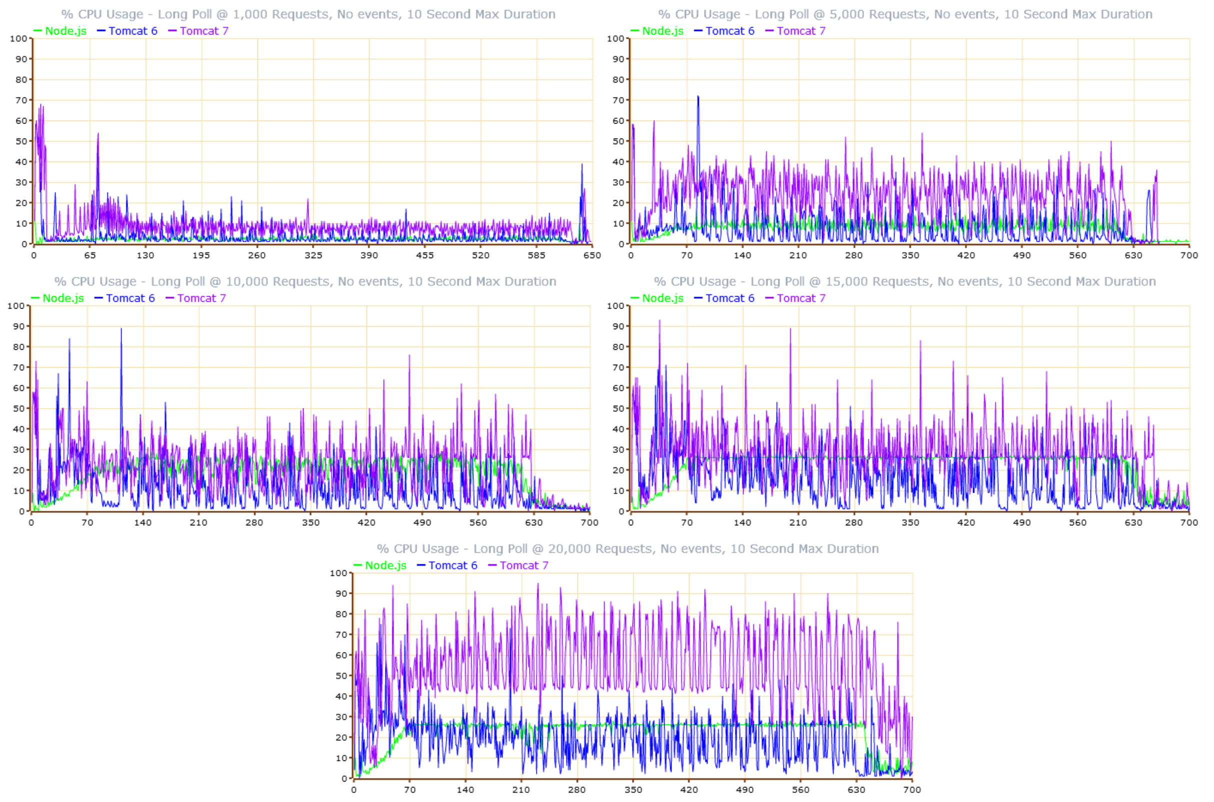


Figure 10: Long Poll – No Events – 10 Second Request Duration – CPU Usage.

As can be seen from the above results, the CPU usage associated with Long Poll requests – even when no events are being emitted – is significantly higher than for Comet (Figure 6). This is to be expected due to the fact that Long Poll requests have the additional overhead of creating and destroying connections for each request – an overhead which Comet does not have.

As the number of concurrent requests increases, the CPU overhead also increases. The most dramatic increase in CPU utilisation is seen with Tomcat 7 which spikes to over 90% CPU usage several times when 20,000 concurrent requests are active. By comparison, Tomcat 6 remains far more consistent across the various numbers of concurrent requests.

As expected, Node.js does not climb above 25% CPU usage – even for 20,000 concurrent requests. This is due to the fact that the tests were carried out against a computer with a quad core CPU. Since Node.js is single threaded, it is only able to make use of one of these cores.

The charts below show the memory usage associated with Long Poll requests at various numbers of concurrent requests. In these charts, the Long Poll requests had a 10 second maximum duration.

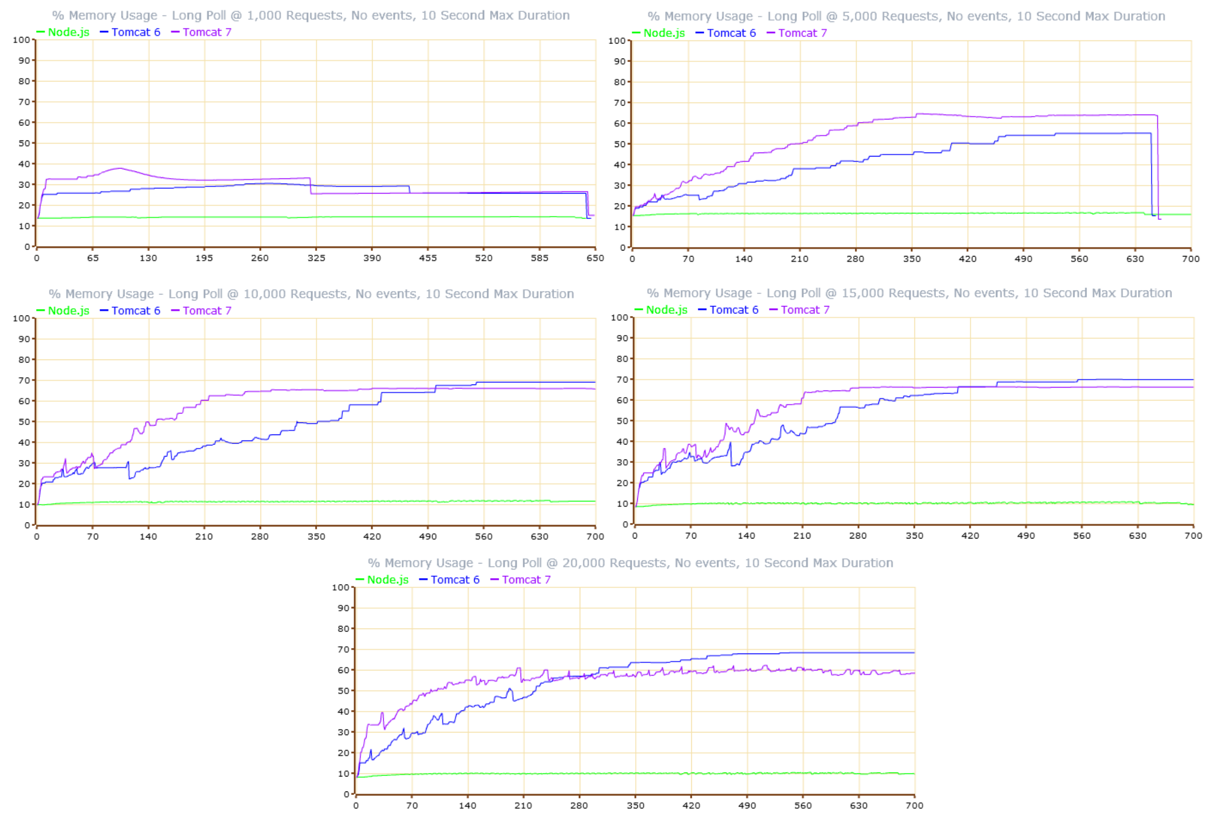


Figure 11: Long Poll – No Events – 10 Second Request Duration – Memory Usage.

As with the CPU utilisation in Figure 10, the memory usage associated with Long Poll requests – even when no events are being emitted – is significantly higher than for Comet (Figure 7). However, in this case, the increased memory usage is only present for the two Tomcat servers – the memory requirements for Node.js remain largely unchanged.

It is interesting to note the differences in the memory usage increase for the two Tomcat servers between Comet and Long Poll. For the Comet tests there was an initial sharp increase in memory usage followed by a plateau. For the Long Poll tests, the increase in memory usage is more gradual. This is likely due to the fact that for Long Poll, requests are constantly being created and destroyed which would have a far more pronounced effect on memory.

The chart below show the number of responses sent for Long Poll requests at various numbers of concurrent requests. In these charts, the Long Poll requests had a 10 second maximum duration:

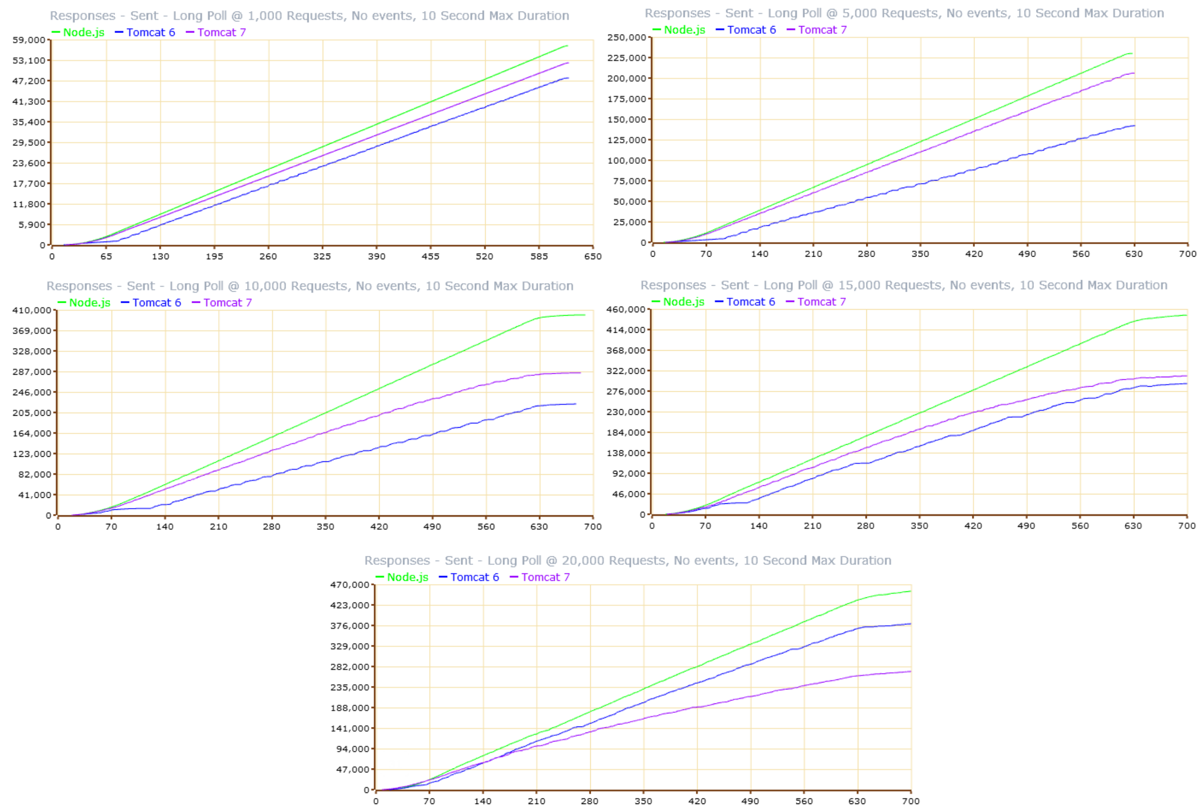


Figure 12: Long Poll – No Events – 10 Second Request Duration – Responses Sent.

As the data above show, Node.js consistently outperformed either of the Tomcat servers with regard to how many responses were sent over the lifetime of the test. It is worth noting (and will be discussed further in section 5.4) that Node.js only outperformed Tomcat when no event emitters were configured, i.e. when the responses being sent were due to a timeout rather than an event being emitted. As the number of concurrent requests increased, the gap in performance between Node.js and Tomcat also increased.

When serving 20,000 concurrent requests, the throughput for each server was as follows (numbers are approximate averages):

Server	Total Response Sent	Responses per second
Node.js	454,000	650
Tomcat 6	380,000	540
Tomcat 7	270,000	385

It is interesting to observe that at lower levels of concurrent requests, Tomcat 7 performed significantly better. For example, at 10,000 concurrent requests, the average responses per second for Tomcat 7 were approximately 440. It is also interesting to note that as the number of concurrent requests increased, a significant increase in the number of errors from Tomcat 7 was recorded.

5.1.4 Long Poll – With Events

In this test category, Long Poll style requests were performed against the various servers. A single event emitter was configured for these tests.

The charts below show the CPU usage associated with Long Poll requests at various numbers of concurrent requests. In these charts, a single event emitter was configured with a frequency of 30 seconds. The Long Poll requests had a 180 second maximum duration to ensure that the event emitter fired before the requests timed out:

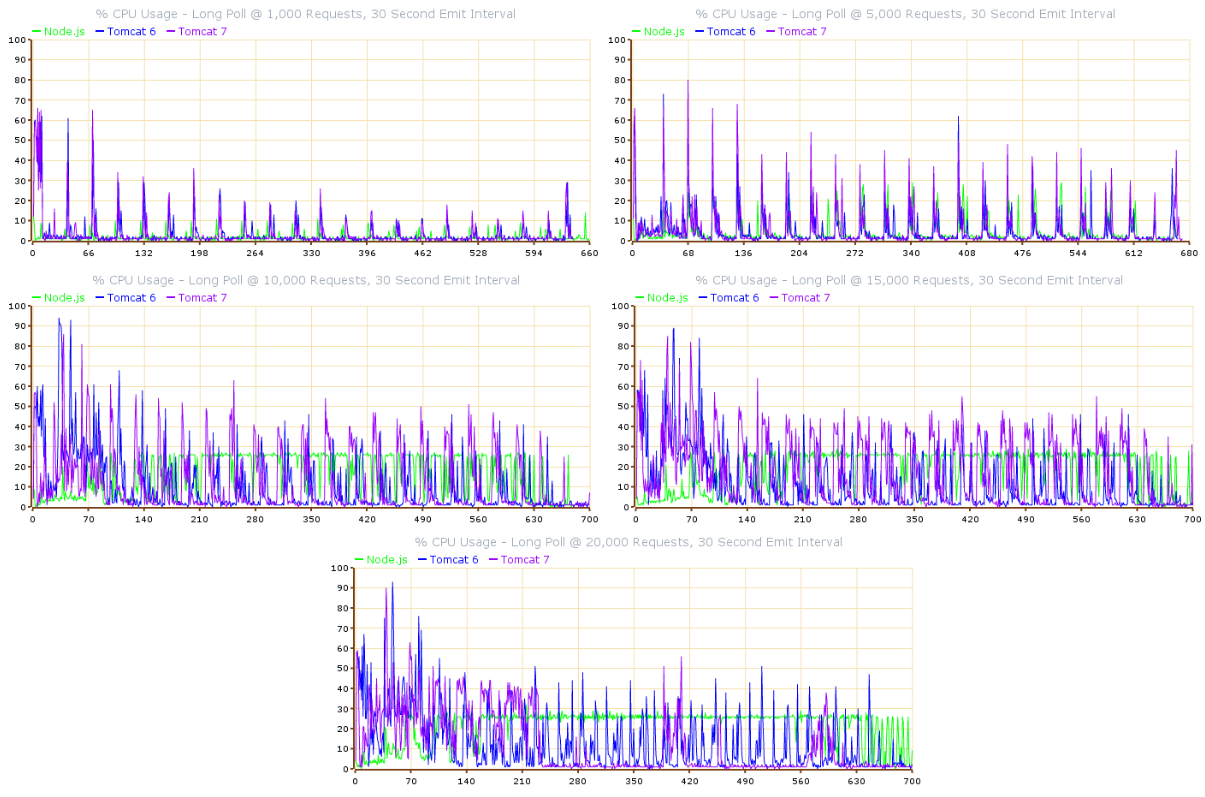


Figure 13: Long Poll – With Events – 30 Second Emit Interval – CPU Usage.

As can be seen from the above samples, CPU usage for Long Poll requests where events are emitted spikes considerably higher than when events are not emitted. It should be noted that the emit interval in this sample is 30 seconds (versus a 10 second Long Poll timeout in the previous section). The reason for using a 30 second emit interval here is that Tomcat 7 frequently crashed with Out Of Memory errors at shorter emit frequencies. In fact, in the chart showing 20,000 concurrent requests, it can be seen that the CPU usage in Tomcat 7 drops to near 0% for much of the test – indicating a server crash. While these crashes were not readily repeatable for every test run, they occurred with sufficient frequency to merit mentioning.

It is interesting to note that CPU usage for Node.js remained largely unchanged for Long Poll regardless of whether events were emitted or not. Typically CPU usage in Node.js reached its 25% maximum around 10,000 concurrent requests regardless of whether event emitters were active.

The charts below show the memory usage associated with Long Poll requests at various numbers of concurrent requests. In the sample below, a single event emitter was configured with a frequency of 30 seconds. The Long Poll requests had a 180 second maximum duration to ensure that the event emitter fired before the requests timed out:

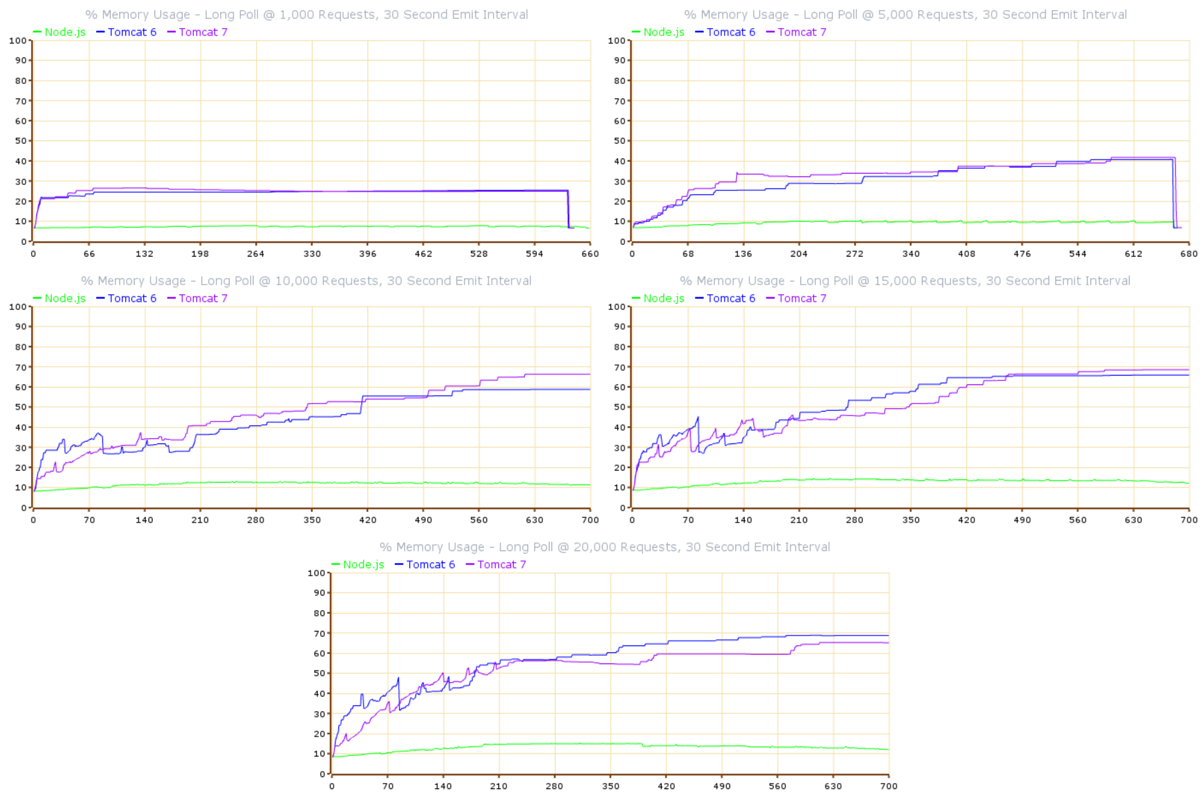


Figure 14: Long Poll – With Events – 30 Second Emit Interval – Memory Usage.

As the above charts show, the memory usage patterns for all three servers are largely consistent with the patterns observed for Long Poll requests when no event emitters were active.

For all tests, both versions of Tomcat were configured with 2,400 MB of available memory (using the Java –Xmx switch). It is unknown why Tomcat 7 repeatedly crashed with Out Of Memory errors when the above charts clearly illustrate that additional memory was available to Tomcat 7 for the 30 second emit interval test (i.e. Tomcat 7 should have been able to consume as much memory as Tomcat 6).

A more detailed investigation of memory usage in Tomcat 7 would be required to establish exactly what was causing the frequent Out Of Memory errors. Unfortunately, this level of investigation was outside the scope of the test performed. This type of detailed memory usage analysis has been noted as possible future work in section 6.

The charts below show the emit duration associated with Long Poll requests at various numbers of concurrent requests. In the sample below, a single event emitter was configured with a frequency of 30 seconds. The Long Poll requests had a 180 second maximum duration to ensure that the event emitter fired before the requests timed out:

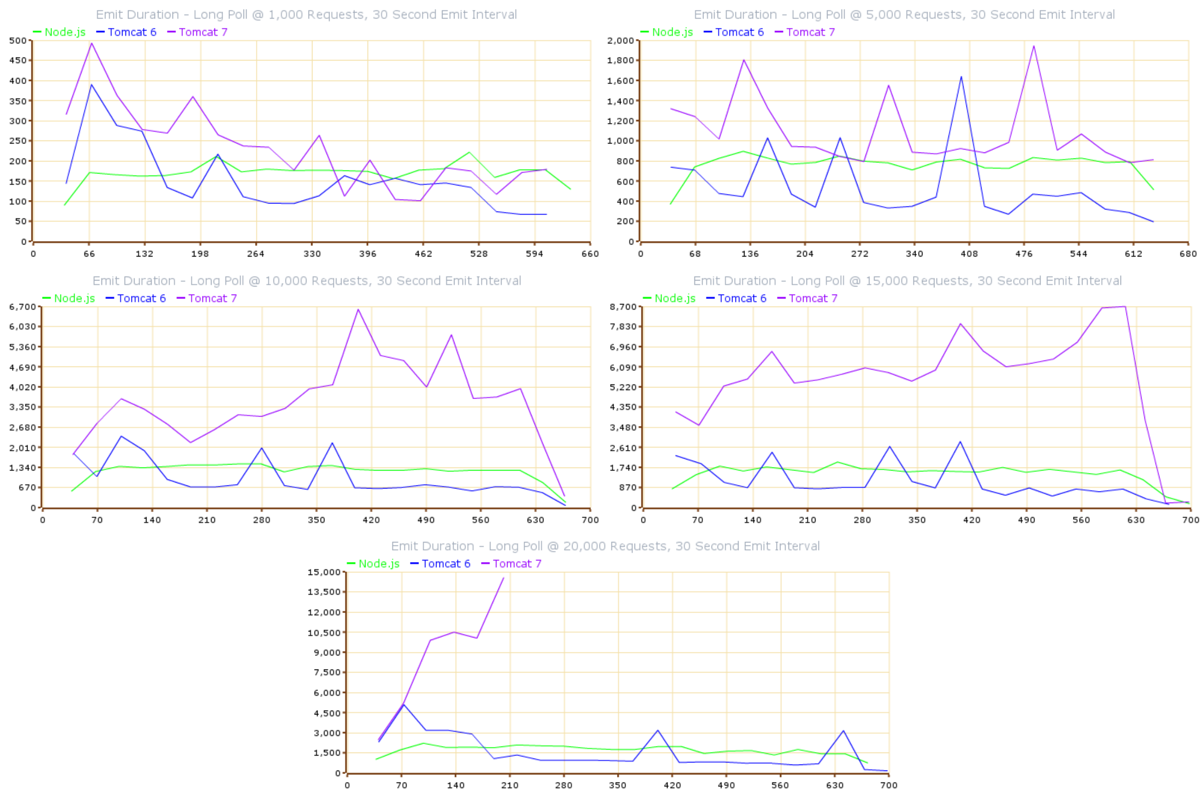


Figure 15: Long Poll – With Events – 30 Second Emit Interval – Varying Requests.

The above charts further reinforce the performance issues noted earlier with Tomcat 7 regarding Long Poll requests with events. In direct contrast to Comet requests with Event Emitters enabled (where Tomcat 7 displayed the best performance), in the above charts we can see that Tomcat 7 performs very poorly when emitting events to Long Poll requests.

It is worth noting that before Tomcat 7 crashed (in the last of the charts above), the emit duration climbed to almost 15 seconds. This indicates that the Out Of Memory errors observed in Tomcat may be related to activity which occurs during an emit loop.

As mentioned earlier, almost all code used by Tomcat 6 and Tomcat 7 was developed in a shared module. The only Tomcat 7 specific code developed was related to the handling of Asynchronous requests. This may well indicate a bug in Tomcat 7. However, extensive searches of the Tomcat 7 user groups and bug tracker did not identify any known issues which could account for the performance issues observed with Long Poll requests.

A very different result set to the one observed in Figure 12 (Long Poll – No Events – 10 Second Request Duration – Responses Sent) is visible when an event emitter is present for Long Poll requests.

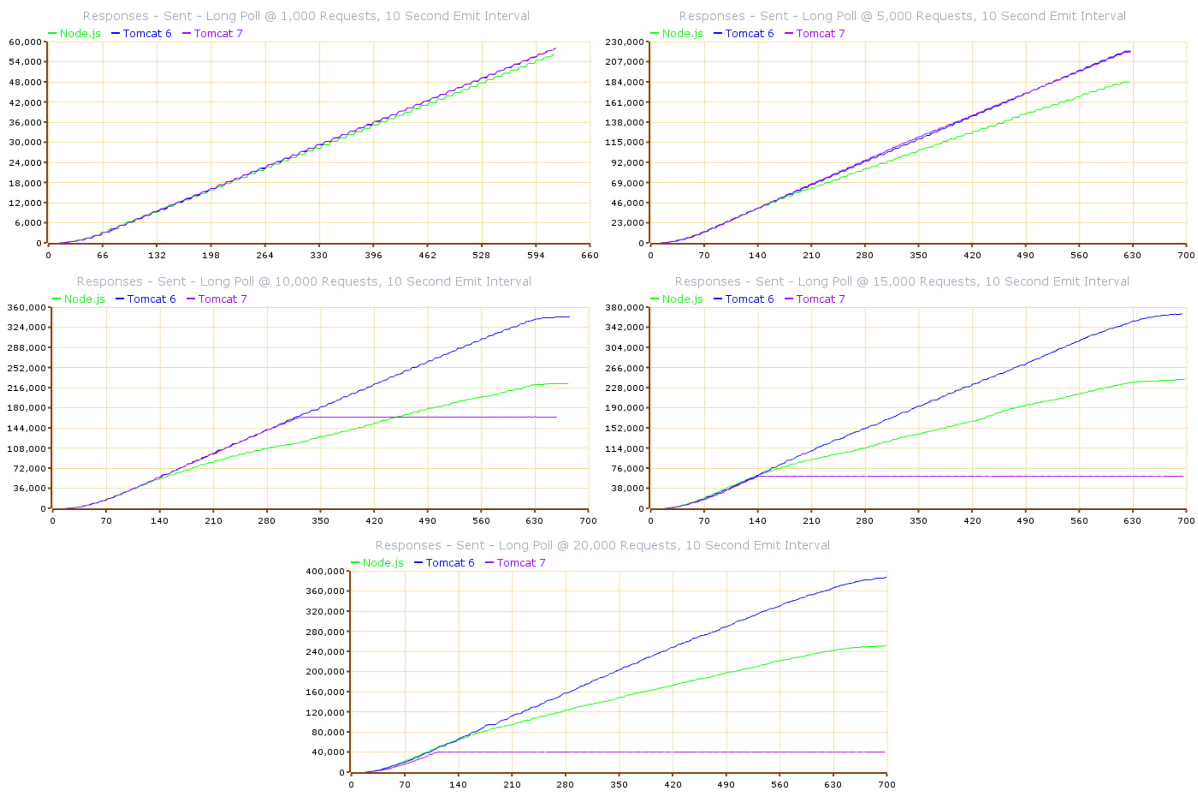


Figure 16: Long Poll – With Events – 10 Second Emit Interval – Responses Sent.

As the results in Figure 16 show, the performance of Node.js changed significantly once event emitters were introduced (compared with the results from Figure 12). The probable cause for this was discussed in section 5.2.2, where it was noted that the single threaded nature of Node.js requires that all overhead associated with sending responses to connected clients must happen on a single thread. By comparison, some of this work can be offloaded to other threads to Tomcat 6 and 7, thereby allowing these servers to complete event loops in a shorter time, and thus serve a higher volume of requests.

These results also demonstrate the performance issues observed with Tomcat 7 for Long Poll style requests. In this case, Tomcat 7 crashed repeatedly when serving 10,000 or more concurrent requests.

The table below lists the throughput for Node.js and Tomcat 6 at 20,000 concurrent connections with a 10 second emit interval (numbers are approximate averages). Note that in the result set sampled, Tomcat 7 crashed very early in the test cycle and so is excluded from the comparison.

Server	Total Response Sent	Responses per second
Node.js	251,000	420
Tomcat 6	387,000	645

5.2 Server Implementation Comparisons

In this section each server implementation (Node.js, Tomcat 6 and Tomcat 7) is reviewed separately under various test conditions.

5.2.1 Node.js

This section focuses specifically on Node.js and analyses its performance under various test conditions.

The charts below compare CPU usage for Comet and Long Poll style requests with a 1 second emit interval across varying concurrent request levels.

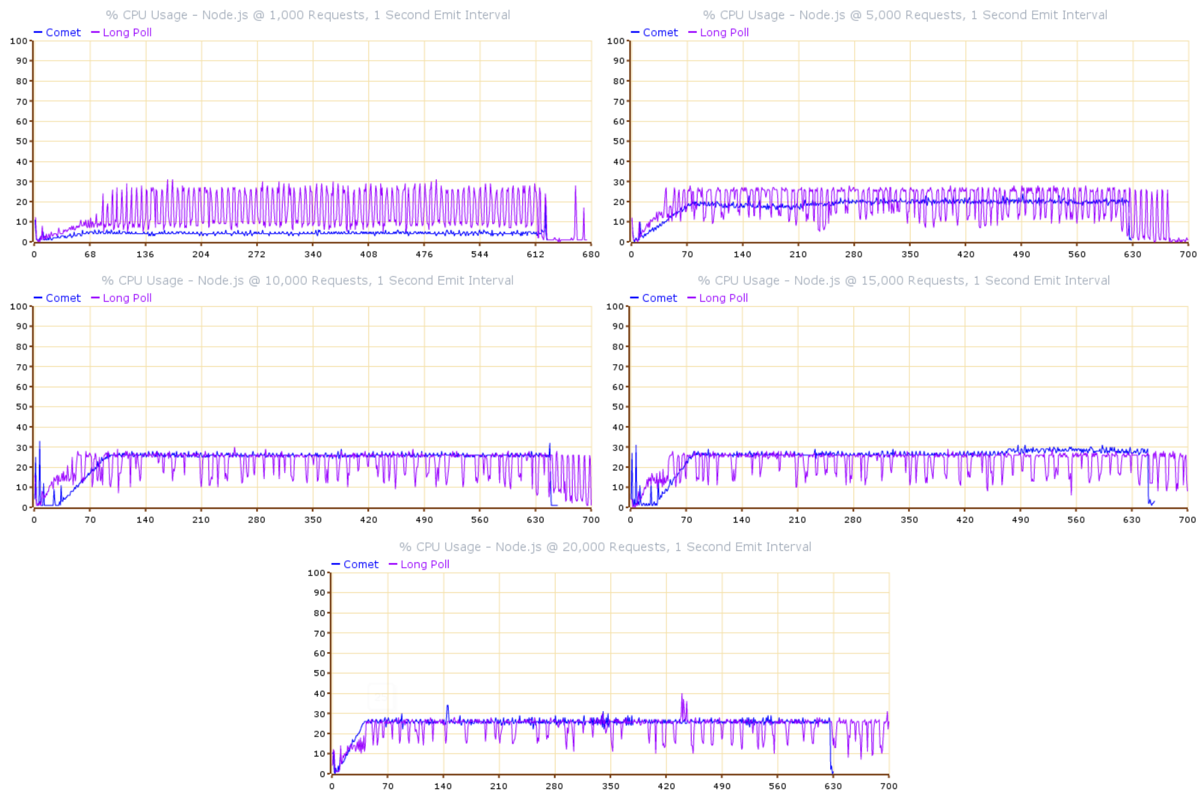


Figure 17: Node.js – Comet vs Long Poll – 1 Second Emit Interval – CPU Usage.

As can be seen from the above samples, there is a significantly higher CPU overhead for Long Poll style requests than Comet requests at low volumes of concurrent connections. However, once the number of concurrent connections rises to approximately 15,000 the CPU usage for Comet style requests consistently remains around the 25% mark (i.e. the maximum available CPU resources for Node.js on a quad core system), while the resource usage for Long Poll style requests tends to fluctuate more.

A significantly different CPU usage pattern can be observed as the emit interval is increased. Figure 18, which shows CPU usage at 20,000 connections with a 30 seconds emit interval, indicates that Comet style requests consume far less CPU cycles than Long Poll at longer emit intervals.

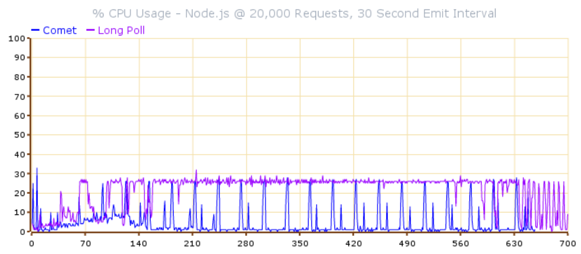


Figure 18: Node.js – Comet vs Long Poll – 30 Second Emit Interval @ 2,0000 connections.

For Long Poll style requests, an interesting metric is the number of responses which can be sent during a test cycle. This value can be influenced by two factors – the frequency with which events are emitted and the number of concurrent requests. Figure 19 below graphs the number of responses sent at varying emit frequencies and concurrent request levels.

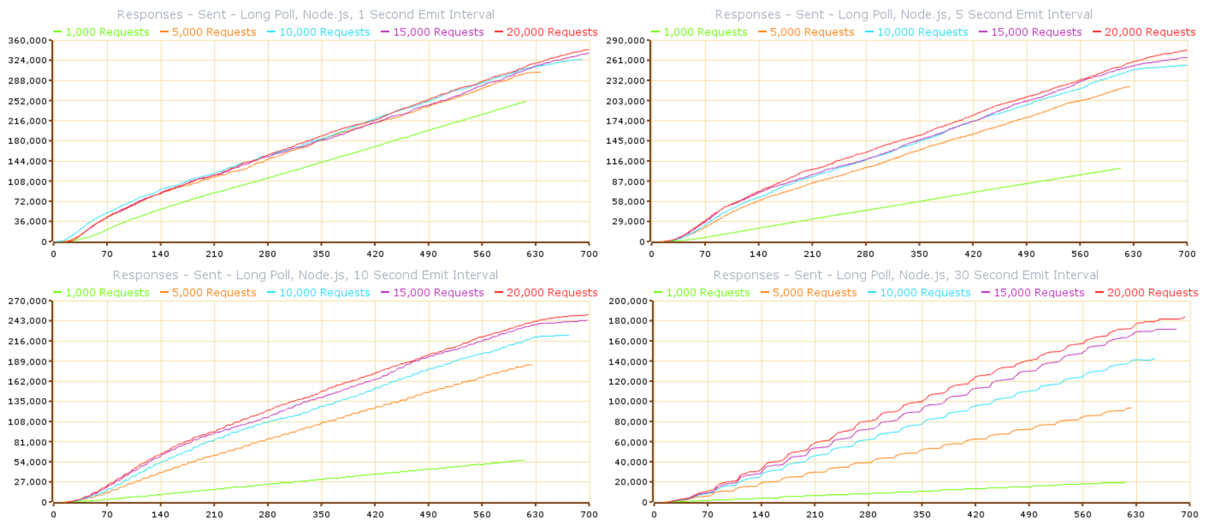
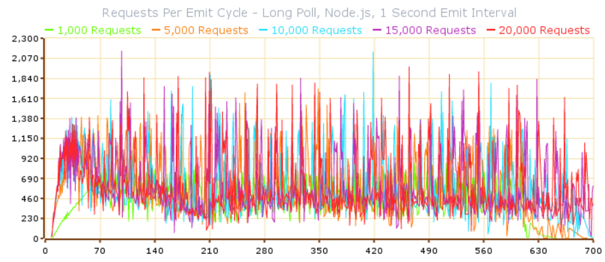


Figure 19: Node.js – Long Poll – Responses Sent – 1 to 30 Second Emit Interval.

The data above indicates that Node.js achieves its highest throughput when the event emit interval is low. However, it should also be noted that Node.js is limited in the number of concurrent requests it can serve at lower emit frequencies, i.e. with a 1 second emit interval there is virtually no difference in the number of request served for 5,000, 10,000, 15,000 and 20,000 concurrent clients. This is due to the fact that the event emitter is firing too frequently to allow more than approximately 5,000 concurrent requests to be served. It is only as the emit interval increases towards 30 seconds that any discernable difference can be observed between the total number of requests served at the various concurrent connection levels.

This observation is further illustrated in Figure 20 which displays the total number of requests served per emit cycle for a 30 seconds emit interval.



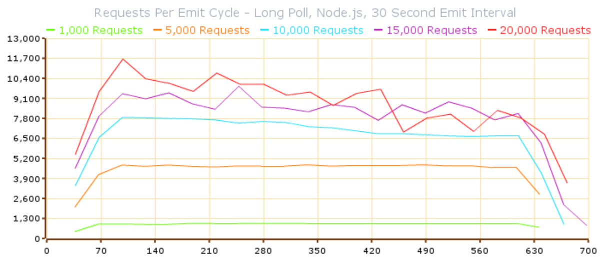


Figure 20: Node.js – Long Poll – Requests per Emit Cycle – 1 Second & 30 Second Emit Intervals.

This data shows that with a 1 second emit interval Node.js manages to serve at most approximately 2,000 concurrent requests – with an average of around 500 requests per emit cycle. Even with a 30 seconds emit interval, Node.js is able to handle at most approximately 12,000 concurrent requests.

5.2.2 Tomcat 6

This section focuses specifically on Tomcat 6 and analyses its performance under various test conditions.

The charts below compare CPU usage for Comet and Long Poll style requests with a 1 second emit interval across varying concurrent request levels.

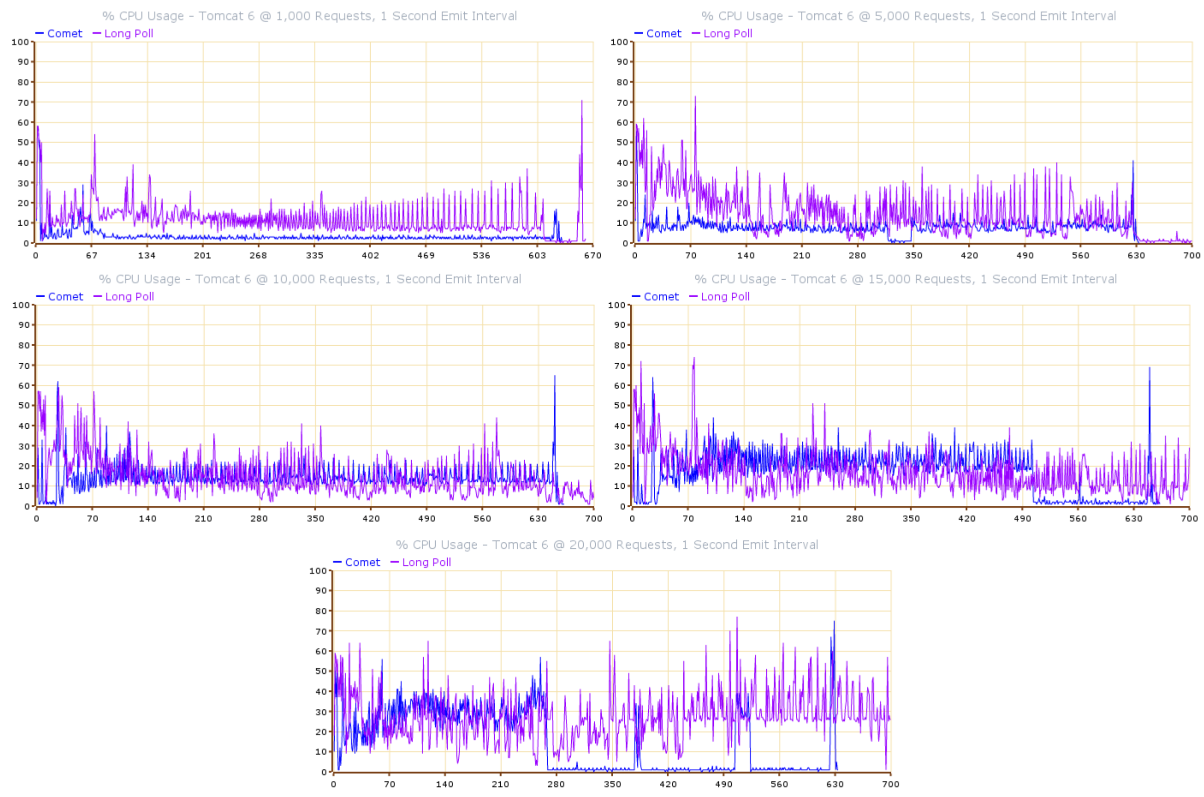


Figure 21: Tomcat 6 – Comet vs Long Poll – 1 Second Emit Interval – CPU Usage.

The above results show a similar CPU usage pattern to the one observed for Node.js in Figure 17 (i.e. Long Poll requests consume more CPU resources than Comet requests). However, this pattern only holds true for Tomcat 6 at lower levels of concurrent connections. As the number of concurrent connections increases, several dips in CPU usage can be observed. This coincides with significant spikes in event emit duration – as show below in Figure 22.

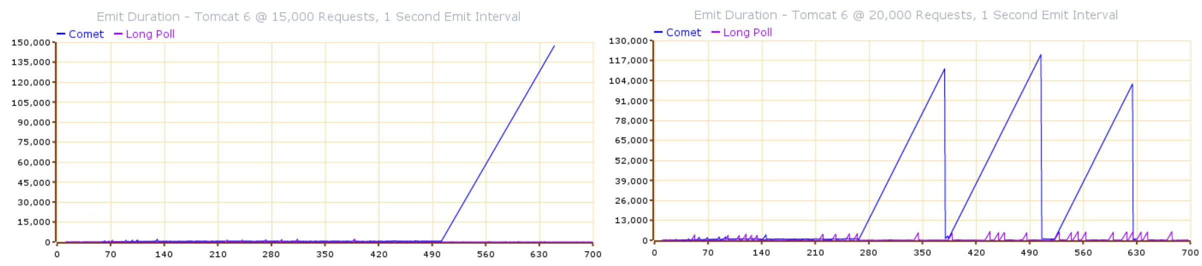


Figure 22: Tomcat 6 – Comet vs Long Poll – 1 Second Emit Interval – Emit Duration.

These spikes in emit duration are similar to the one which was identified with Tomcat 7 in Figure 8. While this pattern of spikes in emit duration was not always readily repeatable, it occurred with significant frequency to be worth noting.

The charts below graph the number of responses sent during a Long Poll test cycle at varying emit intervals and concurrent request levels.

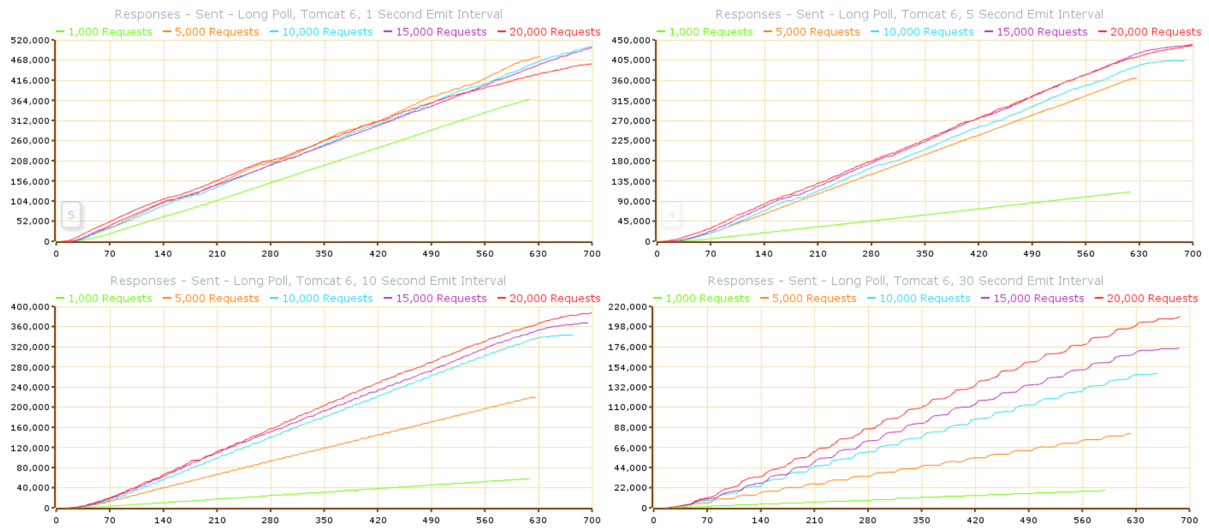


Figure 23: Tomcat 6 – Long Poll – Responses Sent – 1 to 30 Second Emit Interval.

The above charts are the Tomcat 6 variant of the data presented in Figure 19 for Node.js and show a similar performance pattern. Once again, it can be observed that with a 1 second emit interval there is very little difference in the number of requests served regardless of the volume of concurrent connections. As was the case with Node.js, this is due to the fact that such a short emit interval prevents a large number of connections being established between emit cycles.

Figure 24 below further illustrates this point by comparing the number of requests served per emit cycle for a 1 second and 30 second emit interval.

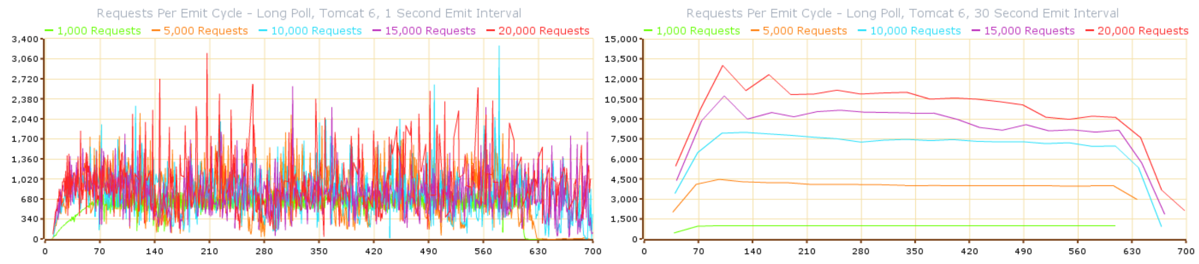


Figure 24: Tomcat 6 – Long Poll – Requests per Emit Cycle – 1 second & 30 second Emit Intervals.

It can be observed from the above results that Tomcat is able to service approximately 800 to 1,000 concurrent connections with an emit interval of 1 second (versus approximately 500 concurrent connections for Node.js). When the emit interval is increased to 30 seconds, Tomcat 6 is able to process approximately 12,500 concurrent requests (which is almost identical to the results observed for Node.js in Figure 20).

5.2.3 Tomcat 7

This section focuses specifically on Node.js and analyses its performance under various test conditions.

The charts below compare CPU usage for Comet and Long Poll style requests with a 1 second emit interval across varying concurrent request levels.

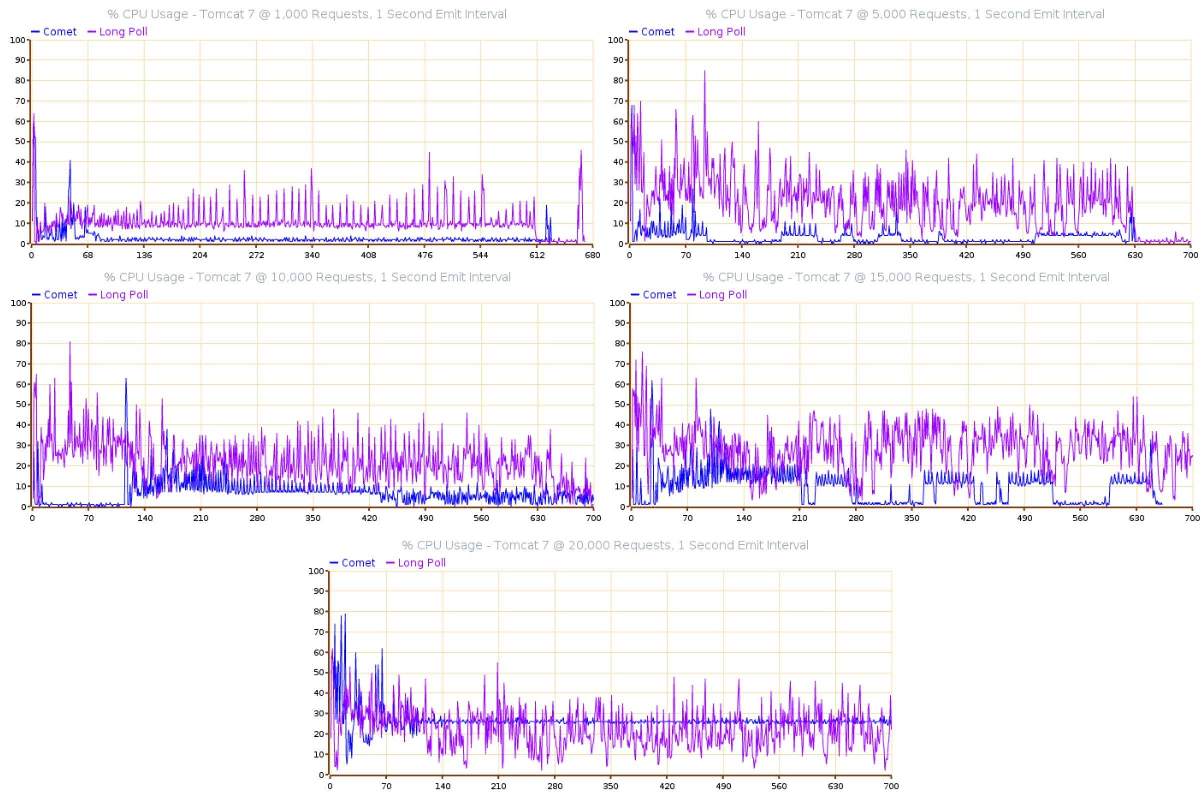


Figure 25: Tomcat 7 – Comet vs Long Poll – 1 Second Emit Interval – CPU Usage.

Tomcat 7 displays some quite erratic CPU usage patterns for Comet requests. While Long Poll CPU usage remained quite consistent – generally increasing as the number of concurrent connections increases – Comet CPU usage patterns fluctuate quite significantly across the various levels of concurrent requests. This pattern of low CPU usage and high emit durations was observed frequently for Tomcat 7, but was not consistent across repeated test runs, and no discernable pattern could be established.

These periods of low (near 0%) CPU usage coincided with periods of increased emit duration – as was the case with Tomcat 6. Figure 26 shows a comparison of emit duration at 5,000 concurrent connections (when CPU usage was fluctuating) and 20,000 concurrent connections (when CPU usage was stable).

While it may appear quite unusual that Tomcat 7 maintained consistent CPU usage levels at 20,000 concurrent requests, but displayed irregular usage patterns at much lover levels of concurrent connections (e.g. 5,000), this was due to the fact that the event emitter failed during the 20,000 concurrent connections test run. This is explored in Figure 26.

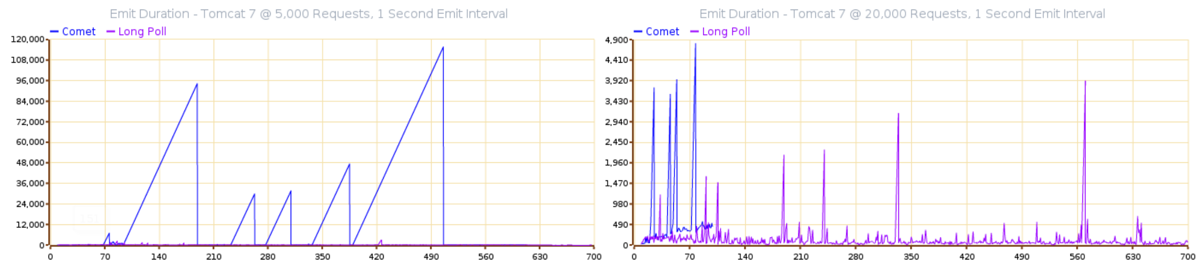


Figure 26: Tomcat 7 – Comet vs Long Poll – 1 Second Emit Interval – Emit Duration.

As the chart on the left in Figure 26 demonstrate, it is possible to clearly identify a correlation between the periods of low CPU and high emit duration. At one point, the emit duration climbs to approximately 110 seconds.

The chart on the right, which shows emit duration at 20,000 concurrent connections, indicates that the event emitter stopped functioning after approximately 90 seconds – notice that the blue line indicating Comet's event emitter frequency stops abruptly at around the 90 second mark.

The charts below graph the number of responses sent during a test cycle at varying emit frequencies and concurrent request levels.

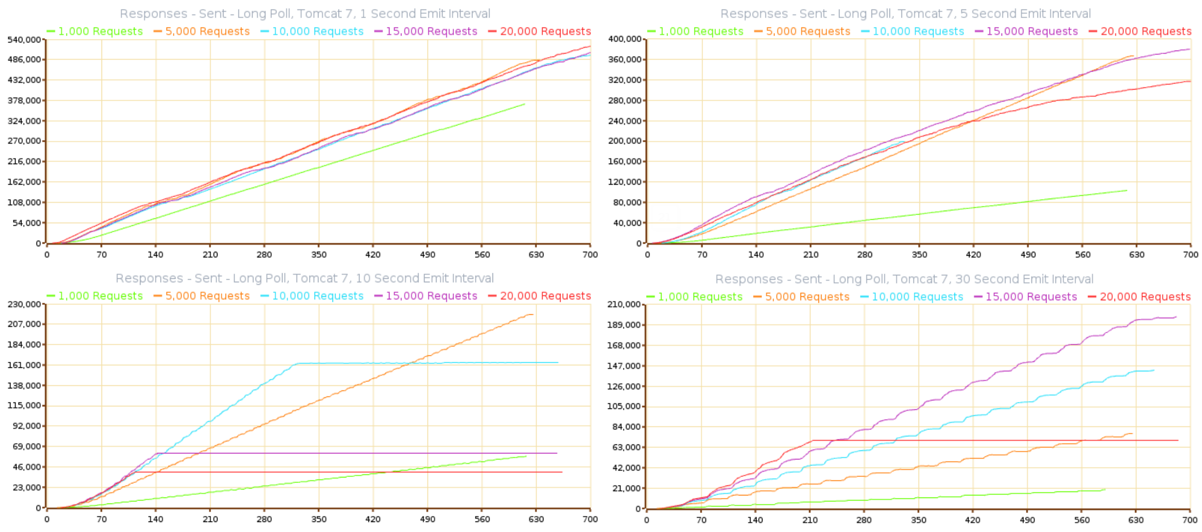


Figure 27: Tomcat 7 – Long Poll – Responses Sent – 1 to 30 Second Emit Interval.

Once again, it is immediately possible to see the inconsistent results produced by Tomcat 7. In this particular set of results we can see that for the 10 second emit interval samples, Tomcat 7 crashed at 10,000, 15,000 and 30,000 concurrent threads. However, at lower emit intervals (1 second and 5 seconds) all samples completed successfully. In fact, at the 1 second emit interval, Tomcat 7 actually outperformed Tomcat 6 with regard to the number of responses sent (521,000 versus 501,000).

6. Conclusion and Future Work

This section provides details of the conclusions drawn based on the findings from section 5 and how these findings relate to the research questions which were originally proposed. It also highlights some potential areas for future work.

6.1 Conclusion

As the results from section 5 shows, a number of common patterns were observed across the entire testing cycle:

- A decrease in the emit interval led to higher CPU usage. This is an expected pattern as a decrease in emit interval means that more events are emitted which puts the servers under higher load.
- An increase in the number of concurrent requests led to increased memory usage for Tomcat servers. This increase was almost linear for Comet style requests, but was somewhat more erratic for Long Poll. Figure 28 (below) provides a summary of the memory usage observed at various levels of concurrent requests for all three servers.

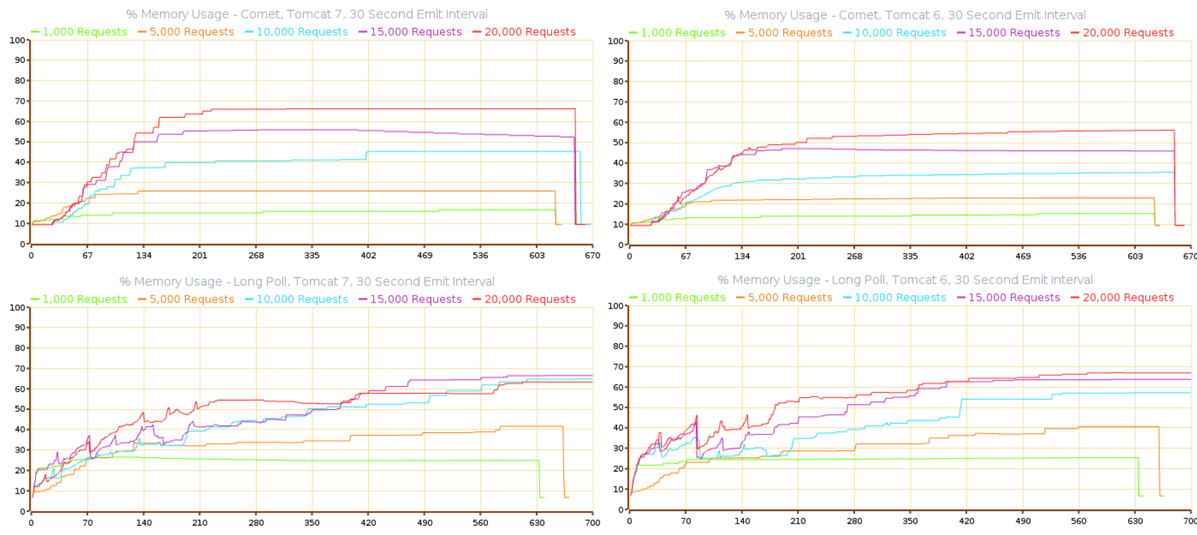


Figure 28: Tomcat 6, Tomcat 7 – 30 Second Emit Interval – Comet, Long Poll – Memory Usage.

As these results demonstrate, Tomcat 7 consistently consumed a higher amount of memory than Tomcat 6. This may well explain the frequent Out Of Memory errors observed with Tomcat 7.

Based on all data gathered, the results point very clearly to Node.js as the best performing server – across all scenarios. Despite the fact that the event emit durations for Node.js were somewhat higher than either of the Tomcat servers, Node.js was by far the most stable server technology, and consumed significantly less CPU and memory in all test casts.

6.2 Research Questions Revisited

The original research proposal posed four questions:

1. Do the characteristics of differing Server Push technologies affect the performance of HTTP Servers?

The results presented in section 5 clearly show significant differences in performance between Comet and Long Poll. Due to the fact that Long Poll requests have the additional overhead of opening and closing HTTP requests for each event emitted, higher levels of both CPU and memory usage were observed across all server types.

2. Can a traditional thread-based web server (Tomcat) be configured to service high volumes of Server Push HTTP requests?

Based on initial testing carried out against Tomcat 6 and Tomcat 7 where standard blocking HTTP connectors were utilised (as described in section 4) rather than non blocking asynchronous connectors, it is clear that the throughput of Tomcat servers can be significantly increased by adopting an asynchronous, non blocking model.

3. Does the Node.js event model perform better than Tomcat's thread based web server when servicing high volumes of Server Push HTTP requests?

All of the data gathered points to the fact that Node.js consistently consumes far less resources than either of the Tomcat servers, and more importantly, Node.js produced a far more stable set of results. In fact, over the course of all testing performed, Node.js did not exhibit any server crashes.

4. Does Node.js provide sufficient functionality to allow it to be configured so that it can seamlessly utilise multiple CPU cores to improve scalability?

Unfortunately this question was not answered over the course of the testing performed, and would make an excellent topic for future work. This is discussed further in section 6.2.3.

6.3 Future Work

The following potential areas of future work were identified over the course of the development and testing.

- **Node.js multi CPU scalability:**

  This topic, which was the subject of one of the proposed research questions, relates to the single threaded nature of Node.js which prevents it from utilising multiple CPU cores. Previous research presented in the Literature Review document identified several promising possibilities for configuring Node.js to launch multiple inter-connected processes which could theoretically work together to utilise multiple CPU cores.

- **Tomcat 7 performance**

  During the course of testing Tomcat 7, several performance issues were observed which resulted in Tomcat 7 frequently crashing. While server crashes were also intermittently observed with Tomcat 6, they were far less frequent. There is significant scope for additional research into performance and memory management in Tomcat 7 with a particular emphasis on changes made to memory management between Tomcat 6 and Tomcat 7.

- **Aggregated results from multiple test runs**

  As mentioned in section 4, due to time constraints, the results presented in section 5 were from a single test run. To verify these results, and produce more accurate data, it would be advisable to re-run the entire test suite several times and generate averaged data from all test runs.

7. References

[1] W3C HTTP Specification - http://www.w3.org/Protocols/rfc1945/rfc1945.

[2] A Russell. "Comet: Low latency data for browsers" Online - http://alex.dojotoolkit.org/wp-content/LowLatencyData.pdf 2006. Last Accessed May 2011.

[3] M McGranaghan "Threaded vs Evented Servers" Online - http://mmcgrana.github.com/2010/07/threaded-vs-evented-servers.html Last Accessed: May 2011.

[4] R. von Behren, J. Condit, and E. Brewer. "Why Events Are A Bad Idea (for High-concurrency Servers)" - 9th Workshop on Hot Topics in Operating Systems (HotOS IX) 2003.

[5] H. C. Lauer and R. M. Needham. "On the duality of operating system structures." In Second International Symposium on Operating Systems, IR1A, October 1978.

[6] J. K. Ousterhout. "Why Threads Are A Bad Idea (for most purposes)" - Presentation given at the 1996 Usenix Annual Technical Conference, January 1996.

[7] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. "Cooperative task management without manual stack management" - 2002 Usenix Annual Technical Conference, June 2002.

[8] N. Repp, R. Berbner, O. Heckmann, and R. Steinmetz. "A crosslayer approach to performance monitoring of web services" In Proceedings of the Workshop on Emerging Web Services Technology. CEUR-WS, Dec 2006.

[9] H. Lucas Jr, "Performance evaluation and monitoring", ACM Computing Surveys, 3(3), Sept. 1971, pp 79-91.

[10] A. Helsinger, R. Lazarus, W. Wright, and J. Zinky. "Tools and Techniques for Performance Measurement of Large Distributed Multiagent Systems" 2nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS), Melbourne, 2003.

[11] L. Griffin, K. Ryan, E. de Leastar and D. Botvic. "Scaling Instant Messaging Communication Services: A Comparison of Blocking and Non-Blocking techniques" In The Sixteenth IEEE symposium on Computers and Communications, 28 June - 1 July 2011, Corfu, Greece. (In Press).

[12] D. Kegel. "The C10k problem" Sept. 2006 Onlne - http://www.kegel.com/c10k.html Last accessed: May 2011.

[13] Tomcat 6.0.0 - Initial Release - http://archive.apache.org/dist/tomcat/tomcat-6/v6.0.0/ - Oct 2006.

[14] Tomcat 6.0.32 - Version used for testing - http://archive.apache.org/dist/tomcat/tomcat-6/v6.0.32/ - Feb 2011.

[15] Java Servlet 3.0 Specification - http://download.oracle.com/auth/otn-pub/jcp/servlet-3.0-fr-eval-oth-JSpec/servlet-3_0-final-spec.pdf - Dec 2009.

[16] Tomcat 7.0.6 - Initial Release - http://archive.apache.org/dist/tomcat/tomcat-7/v7.0.6/ - Jan 2010.

[17] Tomcat 7.0.11 - AsyncContext.setTimeout() bug - https://issues.apache.org/bugzilla/show_bug.cgi?id=51058.

[18] Tomcat 7.0.16 - Version used for testing - http://archive.apache.org/dist/tomcat/tomcat-7/v7.0.16/ - June 2011.

[19] Node.js http request - http://nodejs.org/docs/v0.4.10/api/http.html.

[20] Node.js - Version used for testing - http://nodejs.org/docs/v0.4.10 - July 2011.

[21] BeanShell Assertion - http://jakarta.apache.org/jmeter/usermanual/component_reference.html#BeanShell_Assertion.

[22] JMeter Distributed Mode - http://jakarta.apache.org/jmeter/usermanual/remote-test.html.

[23] Secure Shell (ssh) - http://unixhelp.ed.ac.uk/CGI/man-cgi?ssh+1.

[24] vmstat - http://unixhelp.ed.ac.uk/CGI/man-cgi?vmstat.

[25] tar - http://unixhelp.ed.ac.uk/CGI/man-cgi?tar.

[26] scp - http://unixhelp.ed.ac.uk/CGI/man-cgi?scp.

[27] Cygwin – http://www.cygwin.org.

[28] Open SSH - http://www.openssh.com/.

[29] SSH Public Private Keys - http://www.ece.uci.edu/~chou/ssh-key.html.

[30] Open Flash Charts - http://teethgrinder.co.uk/open-flash-chart-2/.

[31] Java Heap - http://java.sun.com/docs/books/jvms/second_edition/html/Overview.doc.html.

Appendix A – Test Permutations

Server Type	Request Type	Event Emit Frequency (Seconds)	Long Poll Max Duration (Seconds)	Concurrent Requests
Node.js	Comet	1	n/a	1000
Node.js	Comet	1	n/a	5000
Node.js	Comet	1	n/a	10000
Node.js	Comet	1	n/a	15000
Node.js	Comet	1	n/a	20000
Node.js	Comet	5	n/a	1000
Node.js	Comet	5	n/a	5000
Node.js	Comet	5	n/a	10000
Node.js	Comet	5	n/a	15000
Node.js	Comet	5	n/a	20000
Node.js	Comet	10	n/a	1000
Node.js	Comet	10	n/a	5000
Node.js	Comet	10	n/a	10000
Node.js	Comet	10	n/a	15000
Node.js	Comet	10	n/a	20000
Node.js	Comet	30	n/a	1000
Node.js	Comet	30	n/a	5000
Node.js	Comet	30	n/a	10000
Node.js	Comet	30	n/a	15000
Node.js	Comet	30	n/a	20000
Node.js	Comet	n/a	n/a	1000
Node.js	Comet	n/a	n/a	5000
Node.js	Comet	n/a	n/a	10000
Node.js	Comet	n/a	n/a	15000
Node.js	Comet	n/a	n/a	20000
Node.js	Long Poll	1	180	1000
Node.js	Long Poll	1	180	5000
Node.js	Long Poll	1	180	10000
Node.js	Long Poll	1	180	15000
Node.js	Long Poll	1	180	20000
Node.js	Long Poll	5	180	1000
Node.js	Long Poll	5	180	5000
Node.js	Long Poll	5	180	10000
Node.js	Long Poll	5	180	15000
Node.js	Long Poll	5	180	20000
Node.js	Long Poll	10	180	1000
Node.js	Long Poll	10	180	5000
Node.js	Long Poll	10	180	10000
Node.js	Long Poll	10	180	15000

Server Type	Request Type	Event Emit Frequency (Seconds)	Long Poll Max Duration (Seconds)	Concurrent Requests
Node.js	Long Poll	10	180	20000
Node.js	Long Poll	30	180	1000
Node.js	Long Poll	30	180	5000
Node.js	Long Poll	30	180	10000
Node.js	Long Poll	30	180	15000
Node.js	Long Poll	30	180	20000
Node.js	Long Poll	n/a	1	1000
Node.js	Long Poll	n/a	1	5000
Node.js	Long Poll	n/a	1	10000
Node.js	Long Poll	n/a	1	15000
Node.js	Long Poll	n/a	1	20000
Node.js	Long Poll	n/a	5	1000
Node.js	Long Poll	n/a	5	5000
Node.js	Long Poll	n/a	5	10000
Node.js	Long Poll	n/a	5	15000
Node.js	Long Poll	n/a	5	20000
Node.js	Long Poll	n/a	10	1000
Node.js	Long Poll	n/a	10	5000
Node.js	Long Poll	n/a	10	10000
Node.js	Long Poll	n/a	10	15000
Node.js	Long Poll	n/a	10	20000
Node.js	Long Poll	n/a	30	1000
Node.js	Long Poll	n/a	30	5000
Node.js	Long Poll	n/a	30	10000
Node.js	Long Poll	n/a	30	15000
Node.js	Long Poll	n/a	30	20000
Tomcat 6	Comet	1	n/a	1000
Tomcat 6	Comet	1	n/a	5000
Tomcat 6	Comet	1	n/a	10000
Tomcat 6	Comet	1	n/a	15000
Tomcat 6	Comet	1	n/a	20000
Tomcat 6	Comet	5	n/a	1000
Tomcat 6	Comet	5	n/a	5000
Tomcat 6	Comet	5	n/a	10000
Tomcat 6	Comet	5	n/a	15000
Tomcat 6	Comet	5	n/a	20000
Tomcat 6	Comet	10	n/a	1000
Tomcat 6	Comet	10	n/a	5000
Tomcat 6	Comet	10	n/a	10000
Tomcat 6	Comet	10	n/a	15000
Tomcat 6	Comet	10	n/a	20000
Tomcat 6	Comet	30	n/a	1000
Tomcat 6	Comet	30	n/a	5000

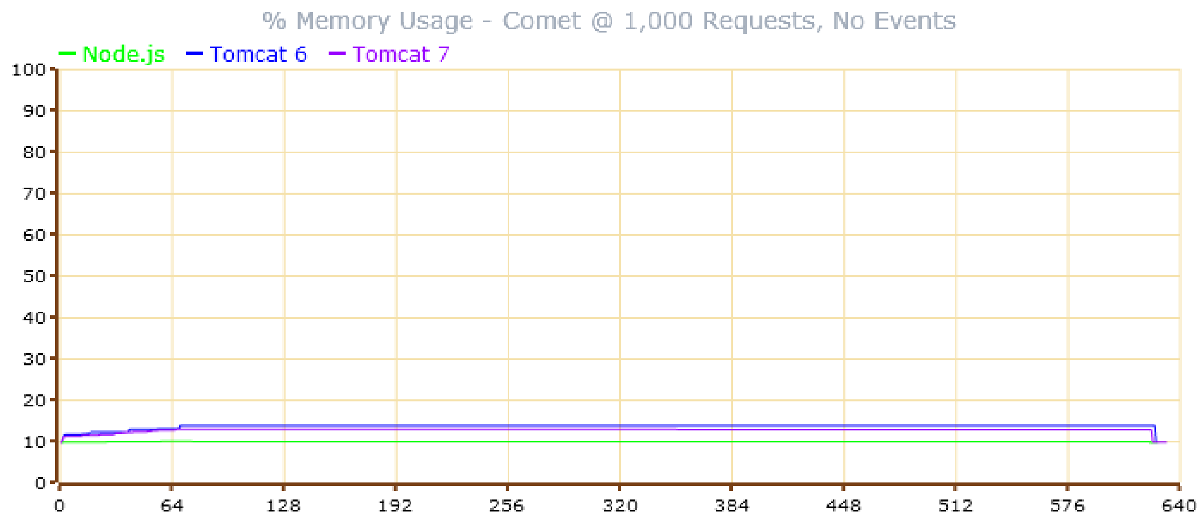
Server Type	Request Type	Event Emit Frequency (Seconds)	Long Poll Max Duration (Seconds)	Concurrent Requests
Tomcat 6	Comet	30	n/a	10000
Tomcat 6	Comet	30	n/a	15000
Tomcat 6	Comet	30	n/a	20000
Tomcat 6	Comet	n/a	n/a	1000
Tomcat 6	Comet	n/a	n/a	5000
Tomcat 6	Comet	n/a	n/a	10000
Tomcat 6	Comet	n/a	n/a	15000
Tomcat 6	Comet	n/a	n/a	20000
Tomcat 6	Long Poll	1	180	1000
Tomcat 6	Long Poll	1	180	5000
Tomcat 6	Long Poll	1	180	10000
Tomcat 6	Long Poll	1	180	15000
Tomcat 6	Long Poll	1	180	20000
Tomcat 6	Long Poll	5	180	1000
Tomcat 6	Long Poll	5	180	5000
Tomcat 6	Long Poll	5	180	10000
Tomcat 6	Long Poll	5	180	15000
Tomcat 6	Long Poll	5	180	20000
Tomcat 6	Long Poll	10	180	1000
Tomcat 6	Long Poll	10	180	5000
Tomcat 6	Long Poll	10	180	10000
Tomcat 6	Long Poll	10	180	15000
Tomcat 6	Long Poll	10	180	20000
Tomcat 6	Long Poll	30	180	1000
Tomcat 6	Long Poll	30	180	5000
Tomcat 6	Long Poll	30	180	10000
Tomcat 6	Long Poll	30	180	15000
Tomcat 6	Long Poll	30	180	20000
Tomcat 6	Long Poll	n/a	1	1000
Tomcat 6	Long Poll	n/a	1	5000
Tomcat 6	Long Poll	n/a	1	10000
Tomcat 6	Long Poll	n/a	1	15000
Tomcat 6	Long Poll	n/a	1	20000
Tomcat 6	Long Poll	n/a	5	1000
Tomcat 6	Long Poll	n/a	5	5000
Tomcat 6	Long Poll	n/a	5	10000
Tomcat 6	Long Poll	n/a	5	15000
Tomcat 6	Long Poll	n/a	5	20000
Tomcat 6	Long Poll	n/a	10	1000
Tomcat 6	Long Poll	n/a	10	5000
Tomcat 6	Long Poll	n/a	10	10000
Tomcat 6	Long Poll	n/a	10	15000
Tomcat 6	Long Poll	n/a	10	20000

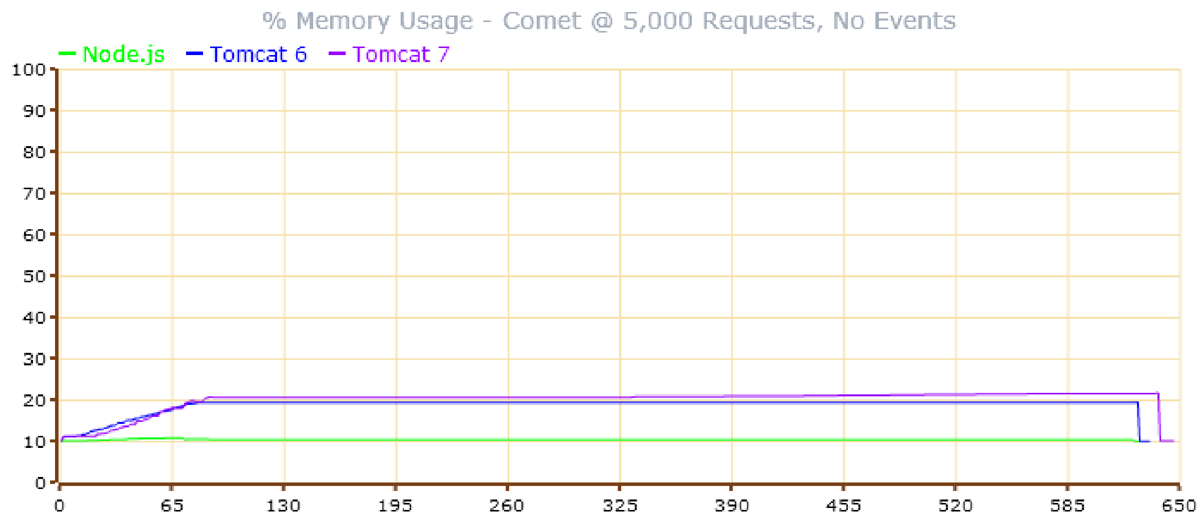
Server Type	Request Type	Event Emit Frequency (Seconds)	Long Poll Max Duration (Seconds)	Concurrent Requests
Tomcat 6	Long Poll	n/a	30	1000
Tomcat 6	Long Poll	n/a	30	5000
Tomcat 6	Long Poll	n/a	30	10000
Tomcat 6	Long Poll	n/a	30	15000
Tomcat 6	Long Poll	n/a	30	20000
Tomcat 7	Comet	1	n/a	1000
Tomcat 7	Comet	1	n/a	5000
Tomcat 7	Comet	1	n/a	10000
Tomcat 7	Comet	1	n/a	15000
Tomcat 7	Comet	1	n/a	20000
Tomcat 7	Comet	5	n/a	1000
Tomcat 7	Comet	5	n/a	5000
Tomcat 7	Comet	5	n/a	10000
Tomcat 7	Comet	5	n/a	15000
Tomcat 7	Comet	5	n/a	20000
Tomcat 7	Comet	10	n/a	1000
Tomcat 7	Comet	10	n/a	5000
Tomcat 7	Comet	10	n/a	10000
Tomcat 7	Comet	10	n/a	15000
Tomcat 7	Comet	10	n/a	20000
Tomcat 7	Comet	30	n/a	1000
Tomcat 7	Comet	30	n/a	5000
Tomcat 7	Comet	30	n/a	10000
Tomcat 7	Comet	30	n/a	15000
Tomcat 7	Comet	30	n/a	20000
Tomcat 7	Comet	n/a	n/a	1000
Tomcat 7	Comet	n/a	n/a	5000
Tomcat 7	Comet	n/a	n/a	10000
Tomcat 7	Comet	n/a	n/a	15000
Tomcat 7	Comet	n/a	n/a	20000
Tomcat 7	Long Poll	1	180	1000
Tomcat 7	Long Poll	1	180	5000
Tomcat 7	Long Poll	1	180	10000
Tomcat 7	Long Poll	1	180	15000
Tomcat 7	Long Poll	1	180	20000
Tomcat 7	Long Poll	5	180	1000
Tomcat 7	Long Poll	5	180	5000
Tomcat 7	Long Poll	5	180	10000
Tomcat 7	Long Poll	5	180	15000
Tomcat 7	Long Poll	5	180	20000
Tomcat 7	Long Poll	10	180	1000
Tomcat 7	Long Poll	10	180	5000
Tomcat 7	Long Poll	10	180	10000

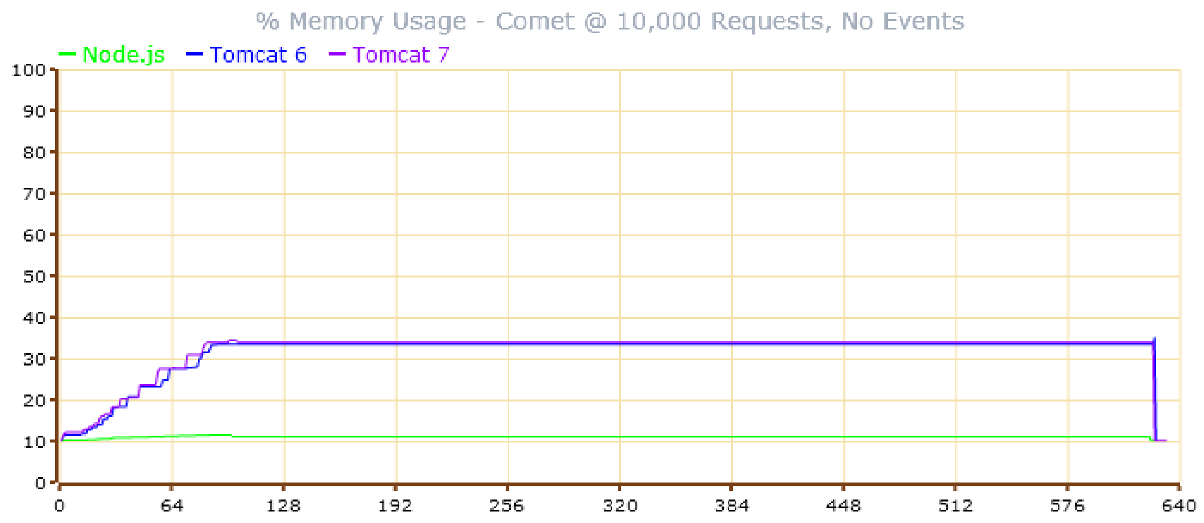
Server Type	Request Type	Event Emit Frequency (Seconds)	Long Poll Max Duration (Seconds)	Concurrent Requests
Tomcat 7	Long Poll	10	180	15000
Tomcat 7	Long Poll	10	180	20000
Tomcat 7	Long Poll	30	180	1000
Tomcat 7	Long Poll	30	180	5000
Tomcat 7	Long Poll	30	180	10000
Tomcat 7	Long Poll	30	180	15000
Tomcat 7	Long Poll	30	180	20000
Tomcat 7	Long Poll	n/a	1	1000
Tomcat 7	Long Poll	n/a	1	5000
Tomcat 7	Long Poll	n/a	1	10000
Tomcat 7	Long Poll	n/a	1	15000
Tomcat 7	Long Poll	n/a	1	20000
Tomcat 7	Long Poll	n/a	5	1000
Tomcat 7	Long Poll	n/a	5	5000
Tomcat 7	Long Poll	n/a	5	10000
Tomcat 7	Long Poll	n/a	5	15000
Tomcat 7	Long Poll	n/a	5	20000
Tomcat 7	Long Poll	n/a	10	1000
Tomcat 7	Long Poll	n/a	10	5000
Tomcat 7	Long Poll	n/a	10	10000
Tomcat 7	Long Poll	n/a	10	15000
Tomcat 7	Long Poll	n/a	10	20000
Tomcat 7	Long Poll	n/a	30	1000
Tomcat 7	Long Poll	n/a	30	5000
Tomcat 7	Long Poll	n/a	30	10000
Tomcat 7	Long Poll	n/a	30	15000
Tomcat 7	Long Poll	n/a	30	20000

Appendix B – Test Result Charts

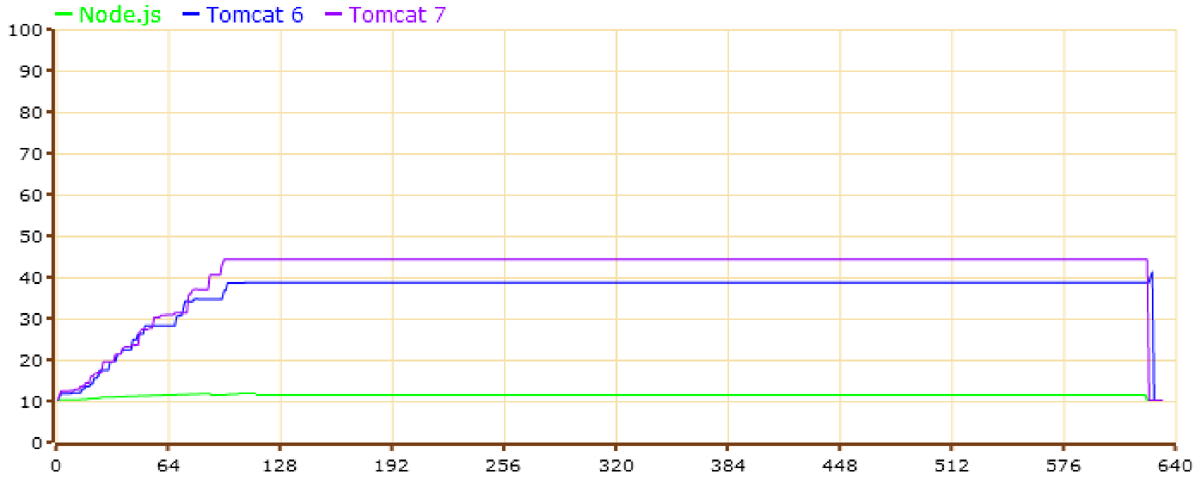
Figure 5 – Comet – No Events – Memory Usage.







% Memory Usage - Comet @ 15,000 Requests, No Events



% Memory Usage - Comet @ 20,000 Requests, No Events

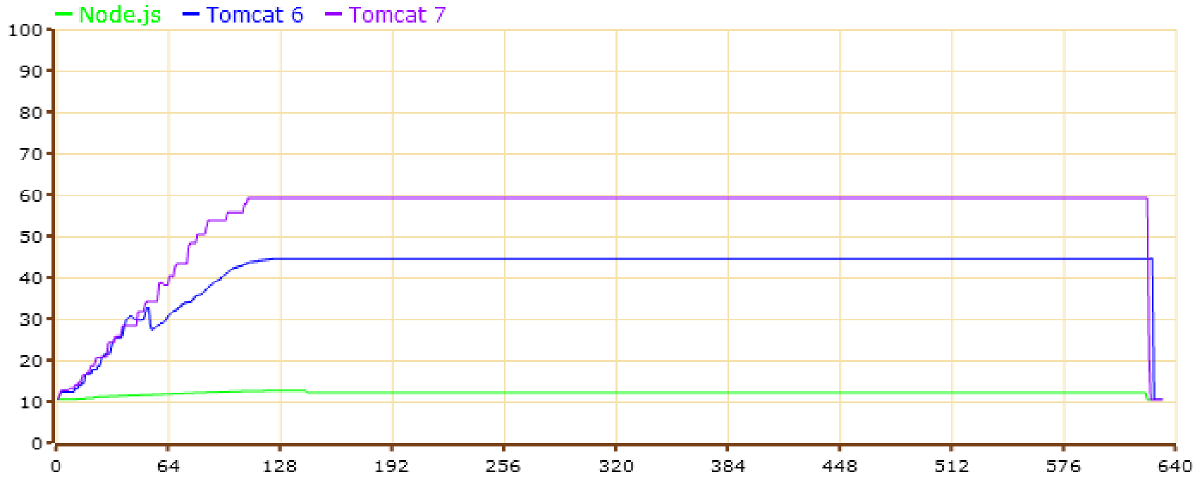
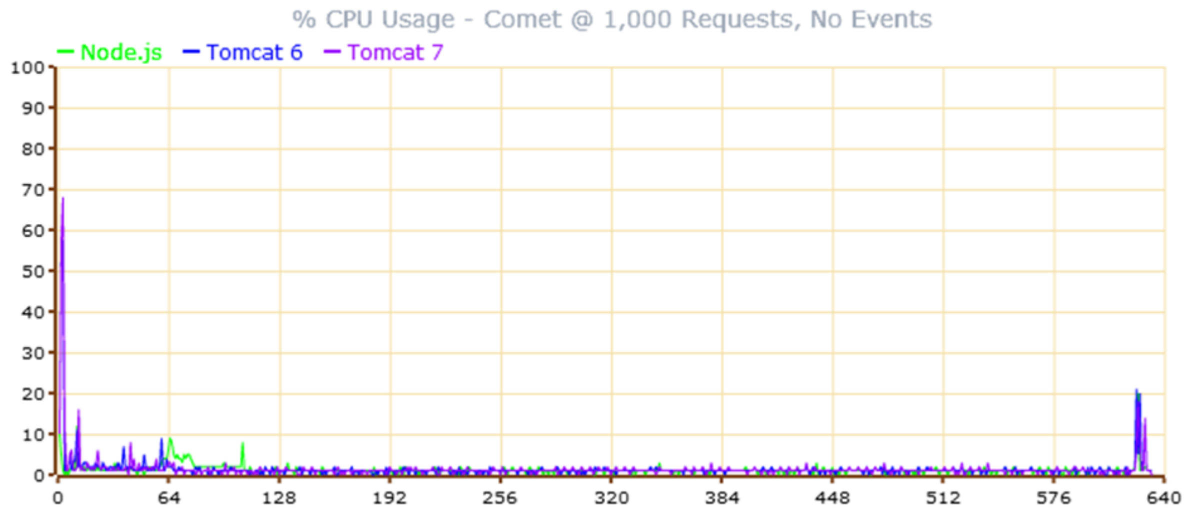
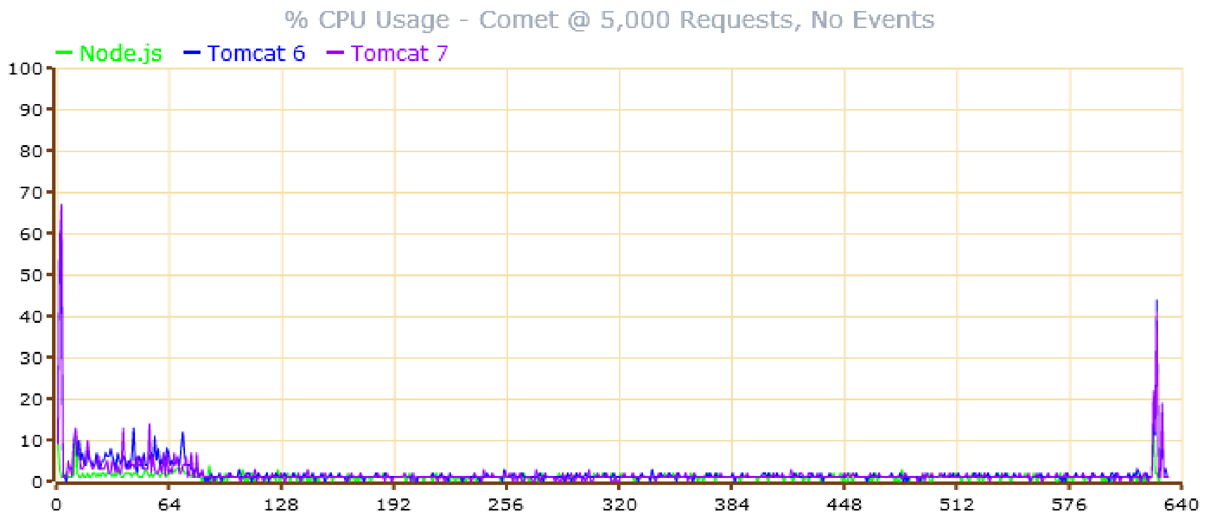
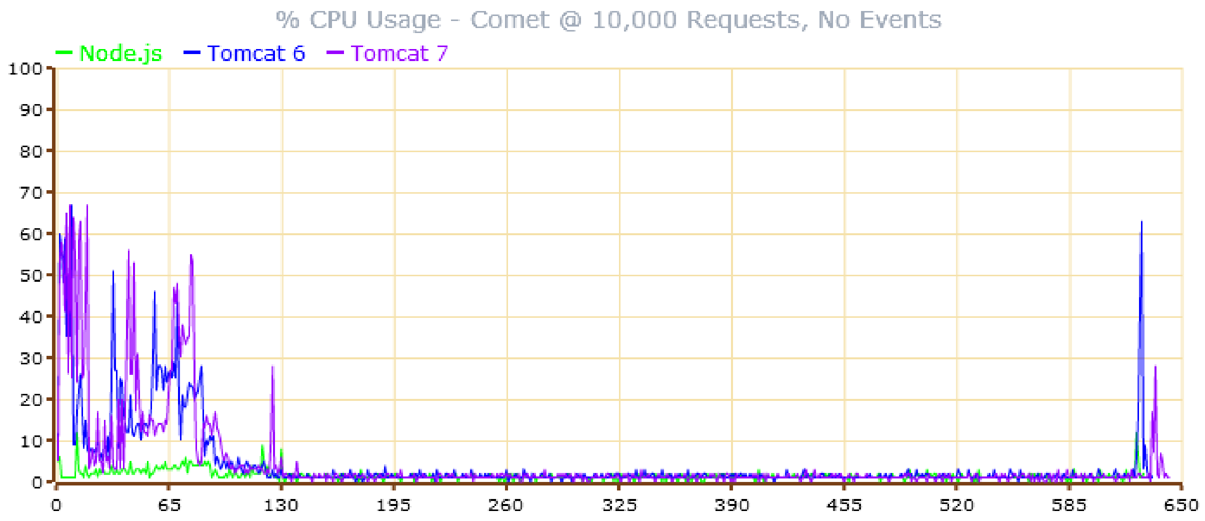
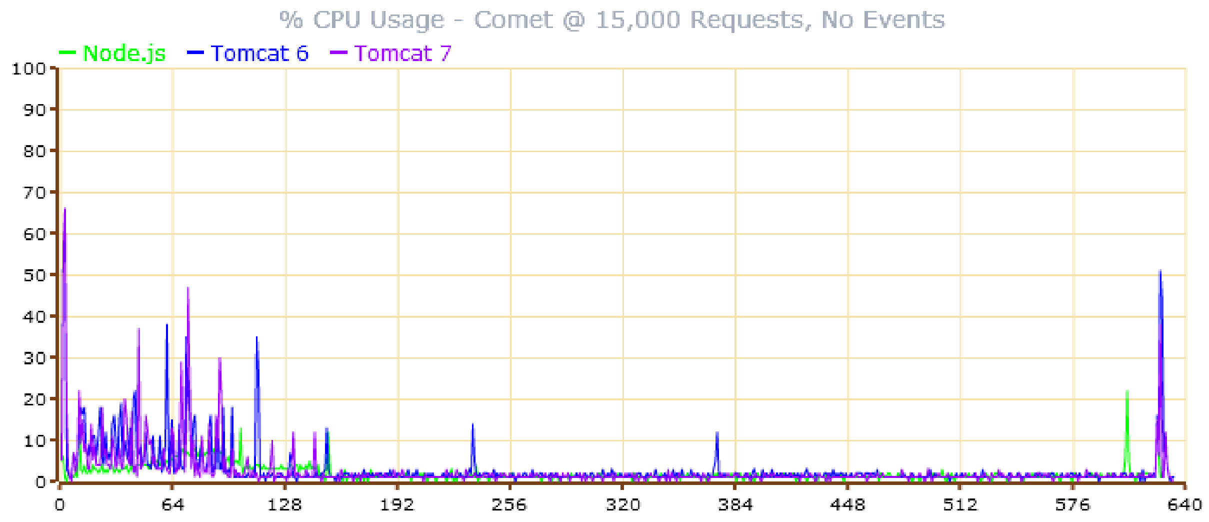


Figure 6 – Comet – No Events – CPU Usage.








% CPU Usage - Comet @ 15,000 Requests, No Events
Node.js
Tomcat 6
Tomcat 7
100
90
80
70
60
50
40
30
20
10
0
0
64
128
192
256
320
384
448
512
576
640

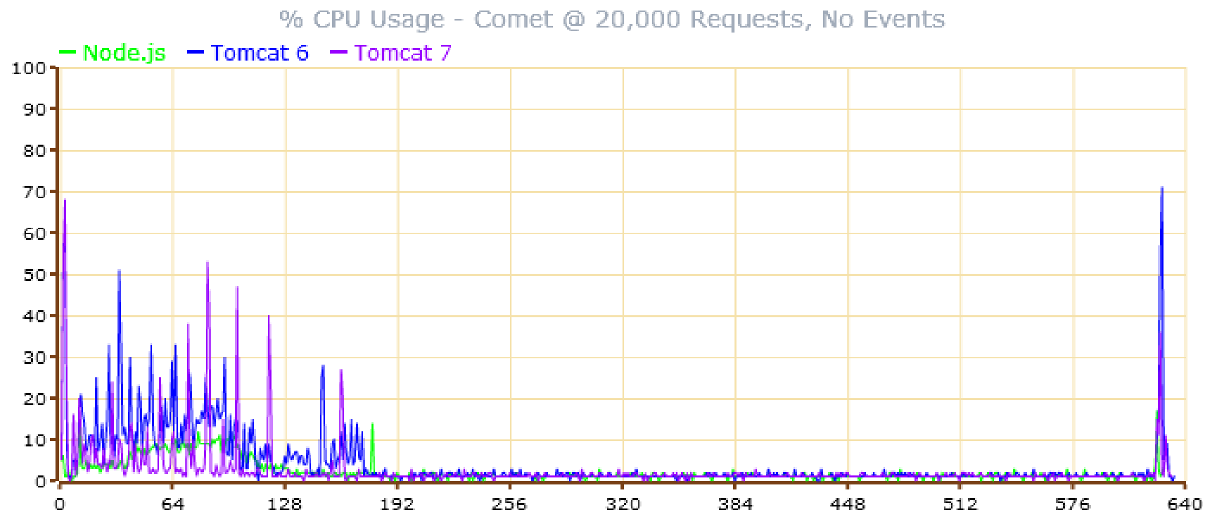
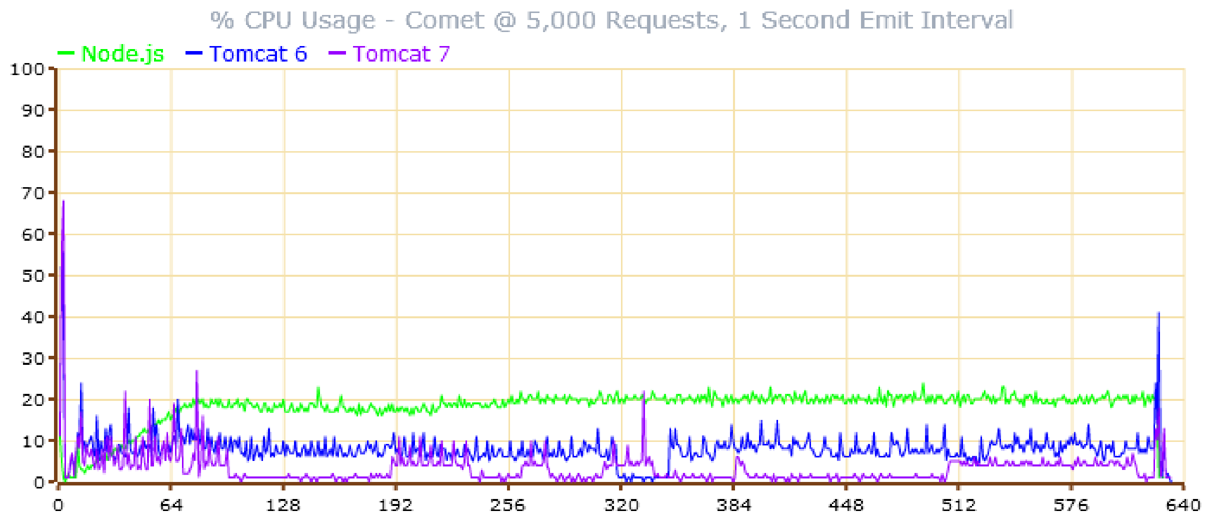
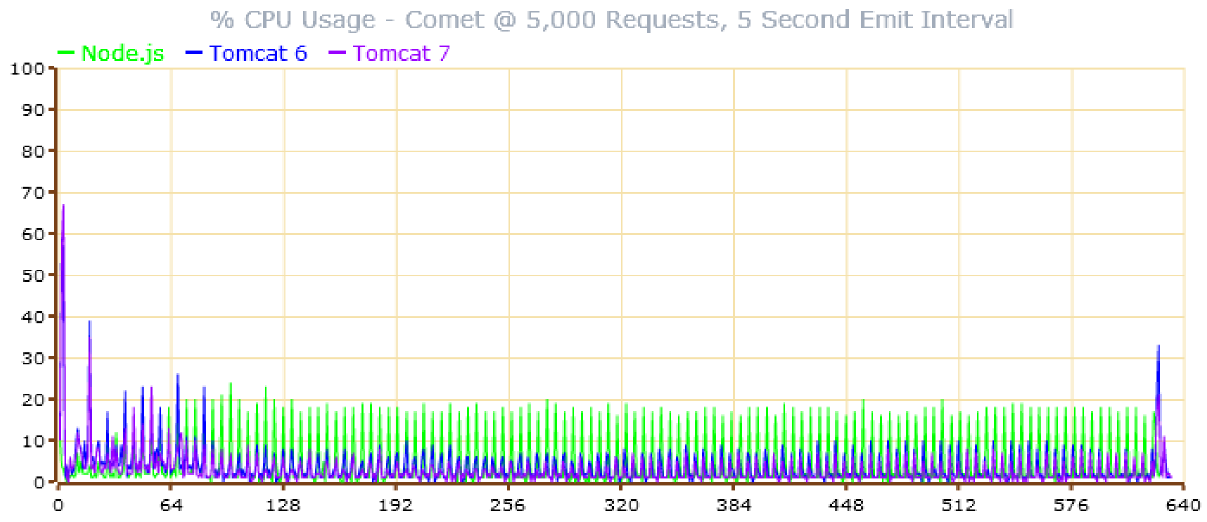
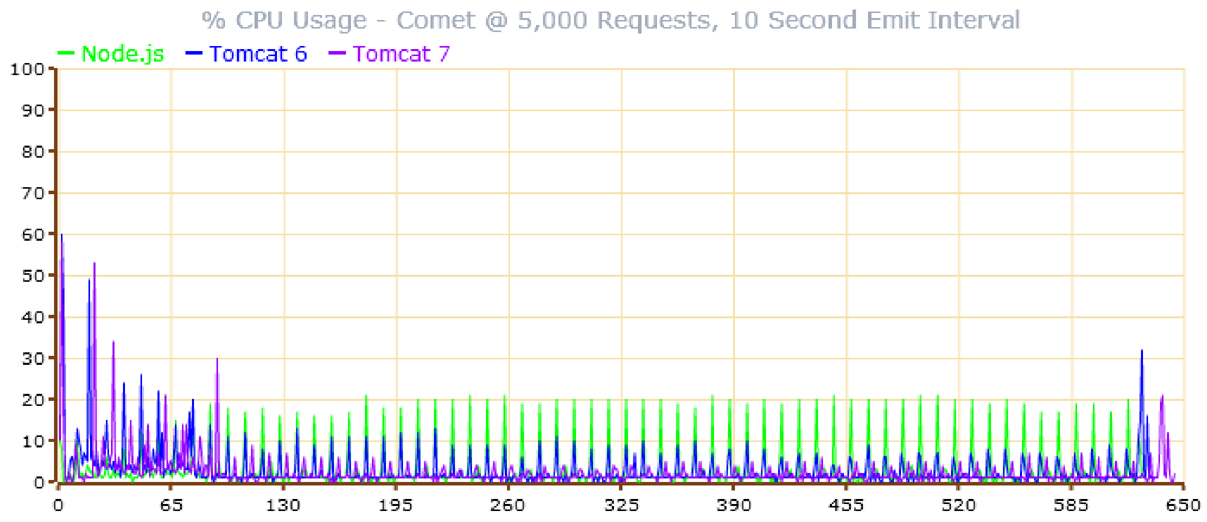
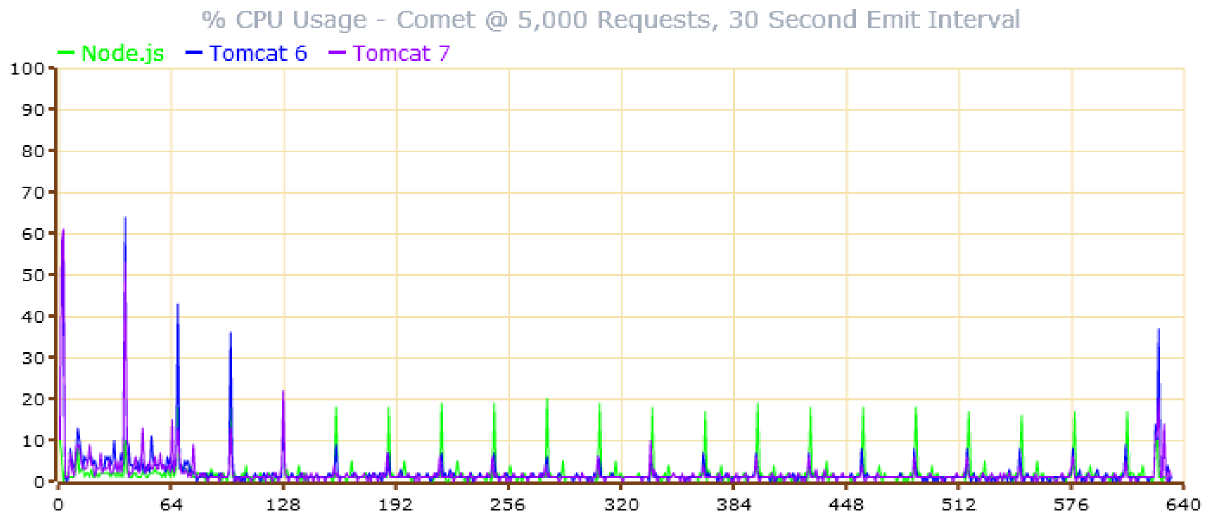

% CPU Usage - Comet @ 20,000 Requests, No Events
Node.js
Tomcat 6
Tomcat 7
100
90
80
70
60
50
40
30
20
10
0
0
64
128
192
256
320
384
448
512
576
640

Figure 7 – Comet – With Events – 10 Second Emit interval – CPU Usage.



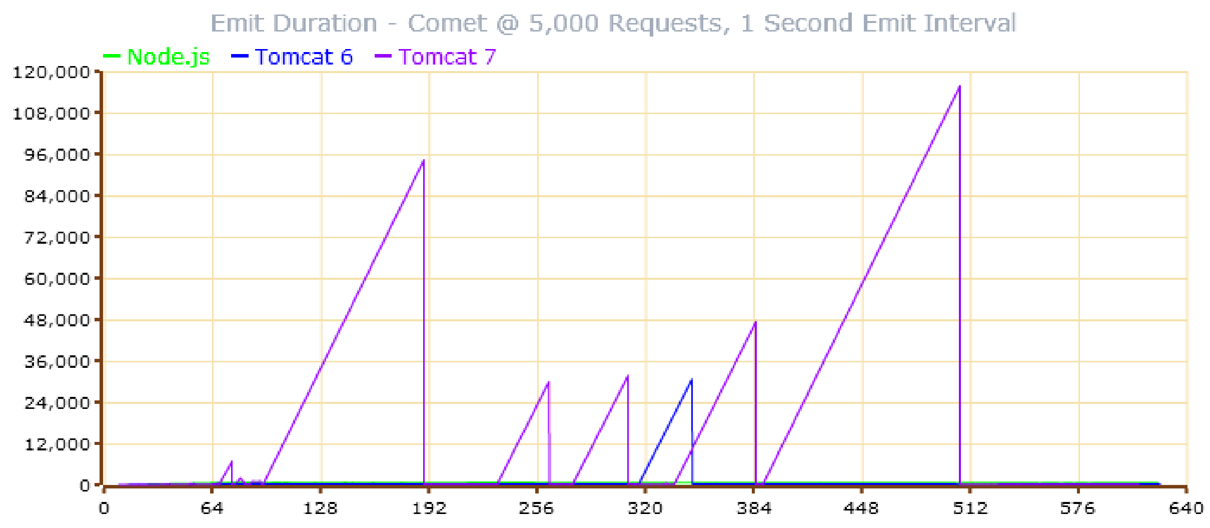


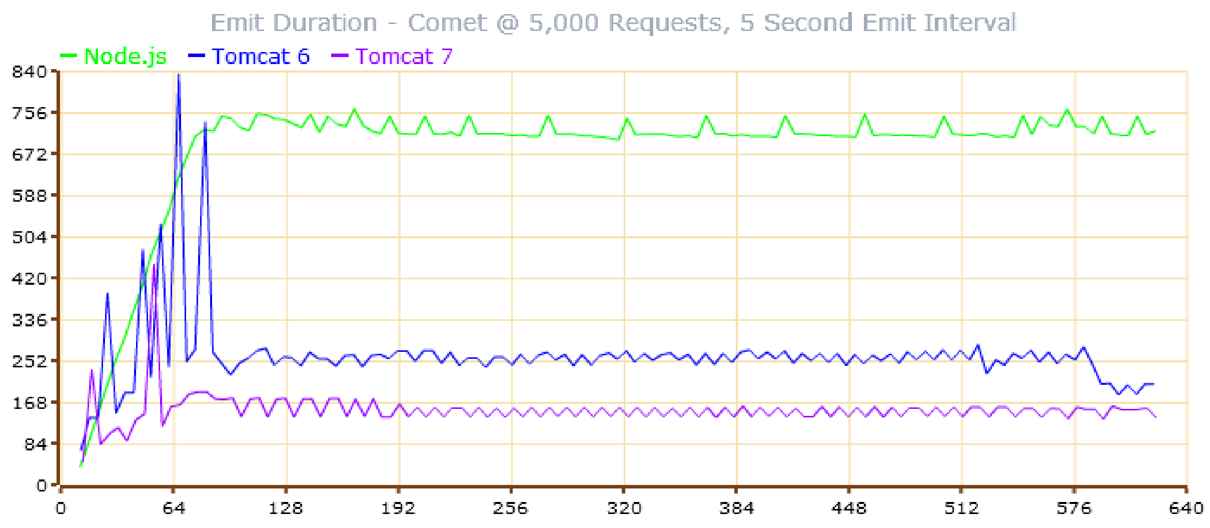


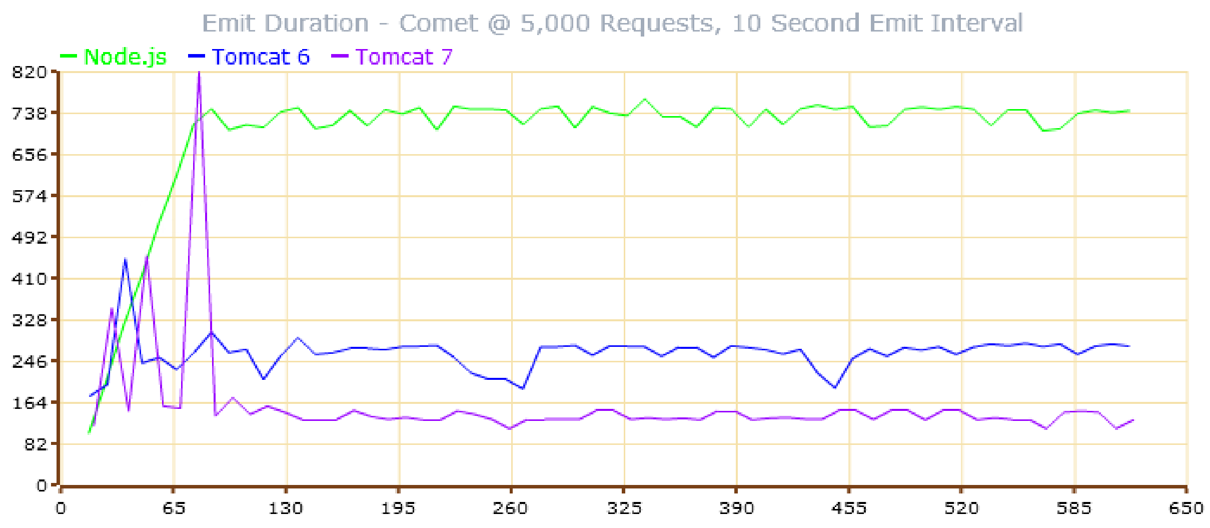


% CPU Usage - Comet @ 5,000 Requests, 30 Second Emit Interval
Node.js
Tomcat 6
Tomcat 7
100
90
80
70
60
50
40
30
20
10
0
0
64
128
192
256
320
384
448
512
576
640

Figure 8 – Comet – With Events – 5,000 Requests – Varying Emit Duration.







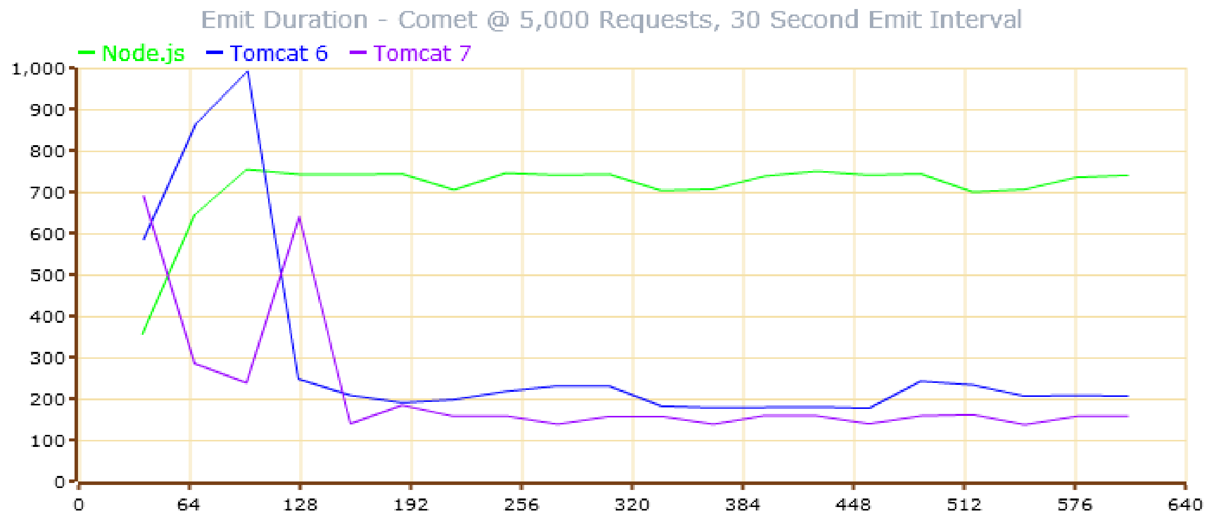
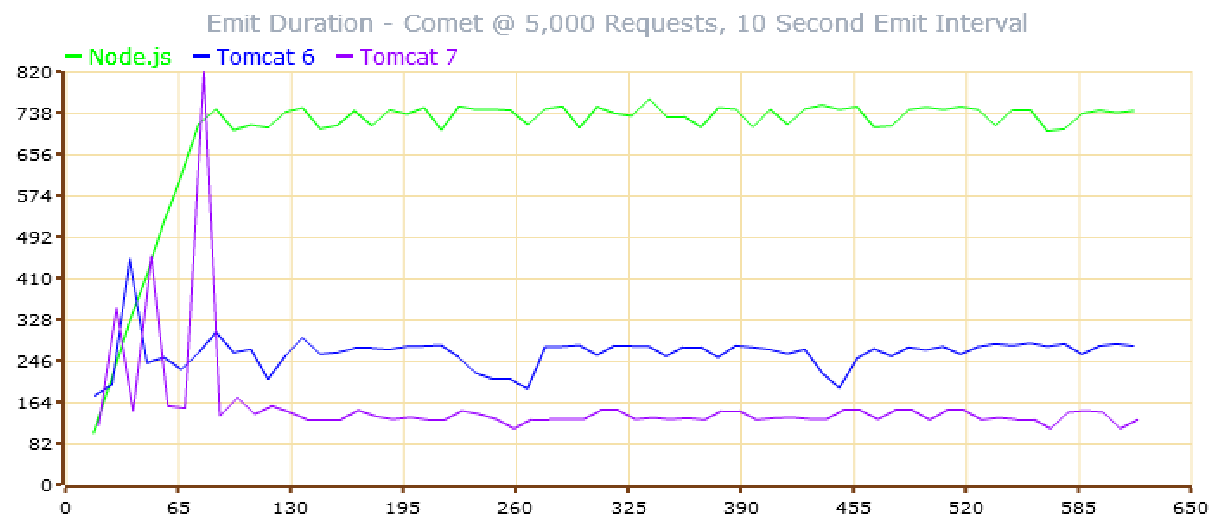
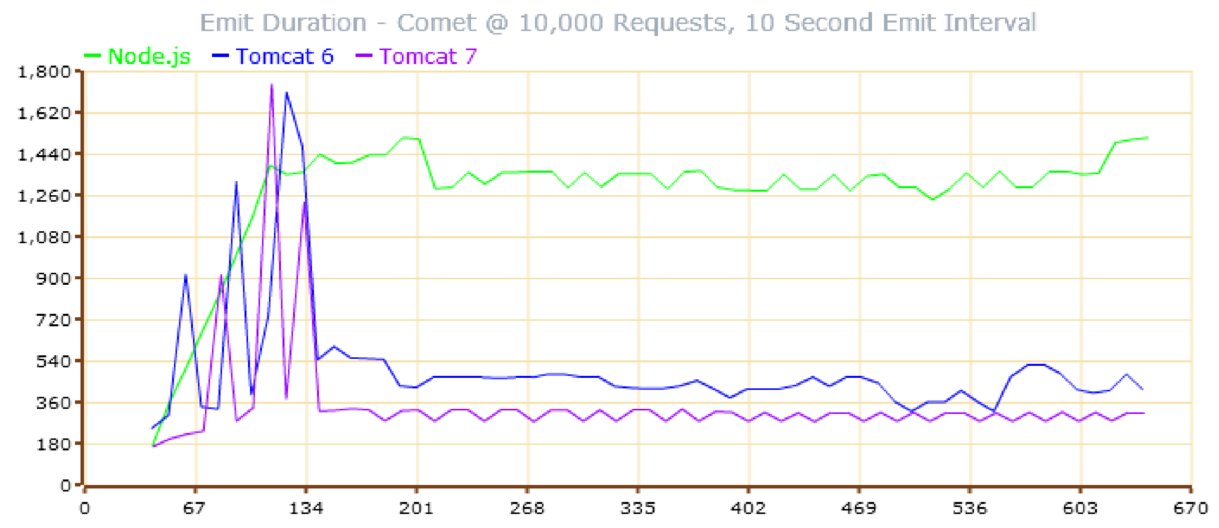
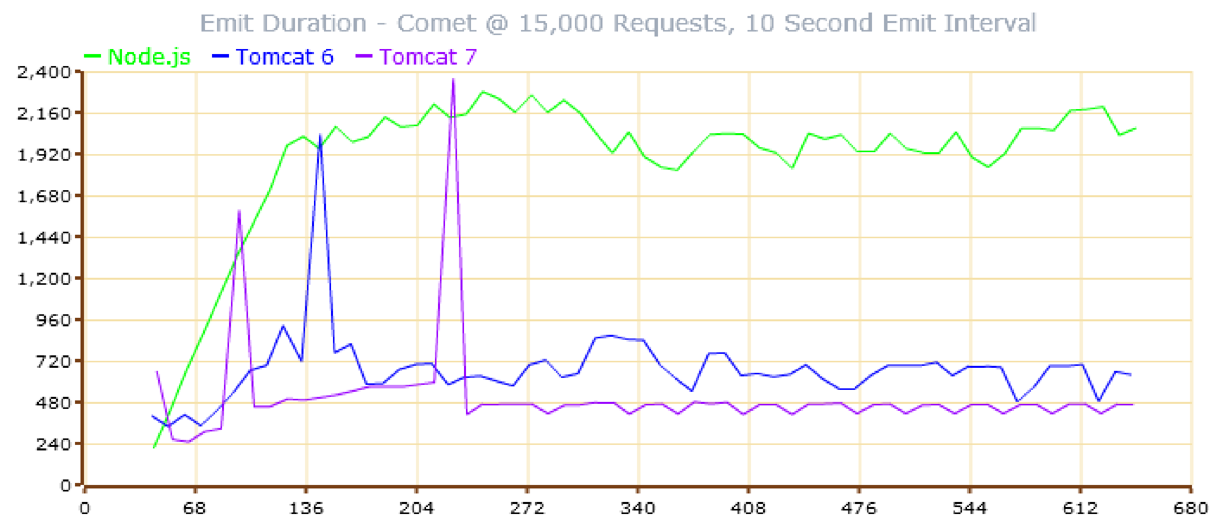

Emit Duration - Comet @ 5,000 Requests, 30 Second Emit Interval
Node.js
Tomcat 6
Tomcat 7
1,000
900
800
700
600
500
400
300
200
100
0
0
64
128
192
256
320
384
448
512
576
640

Figure 9 – Comet – With Events – 10 Second Emit interval – Varying Requests.







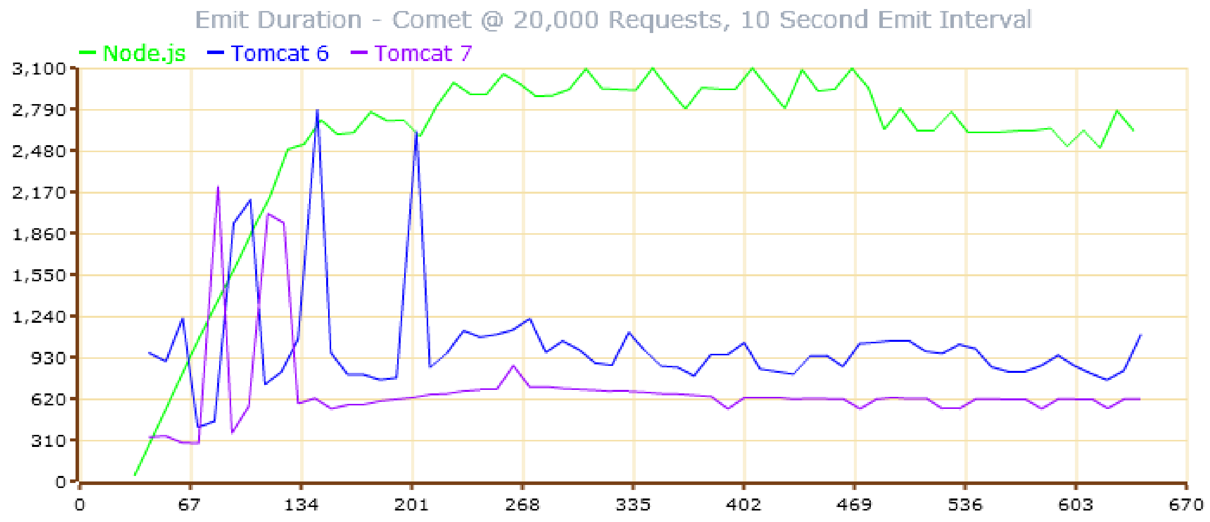
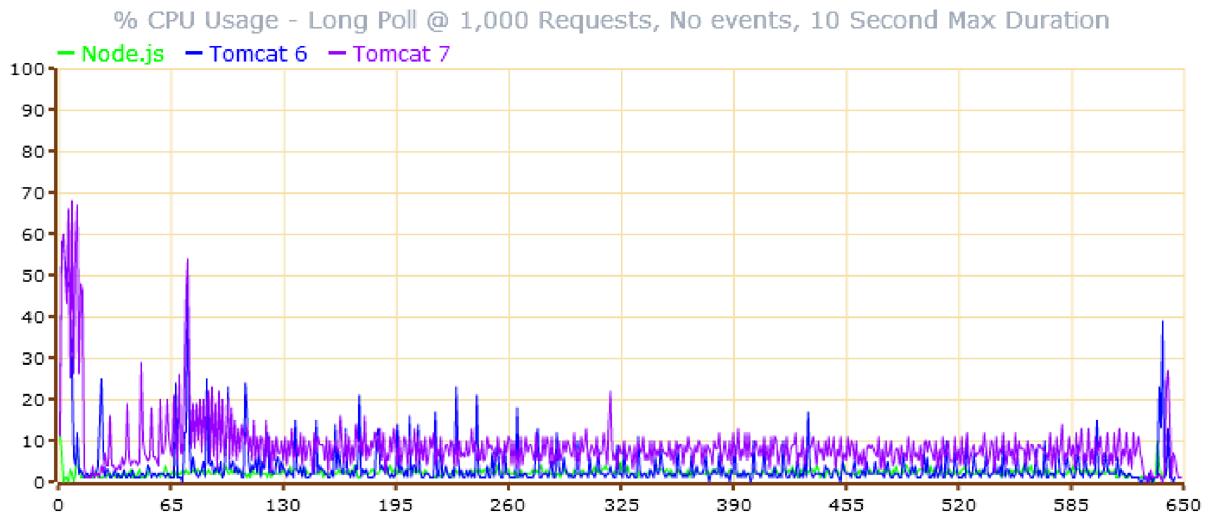
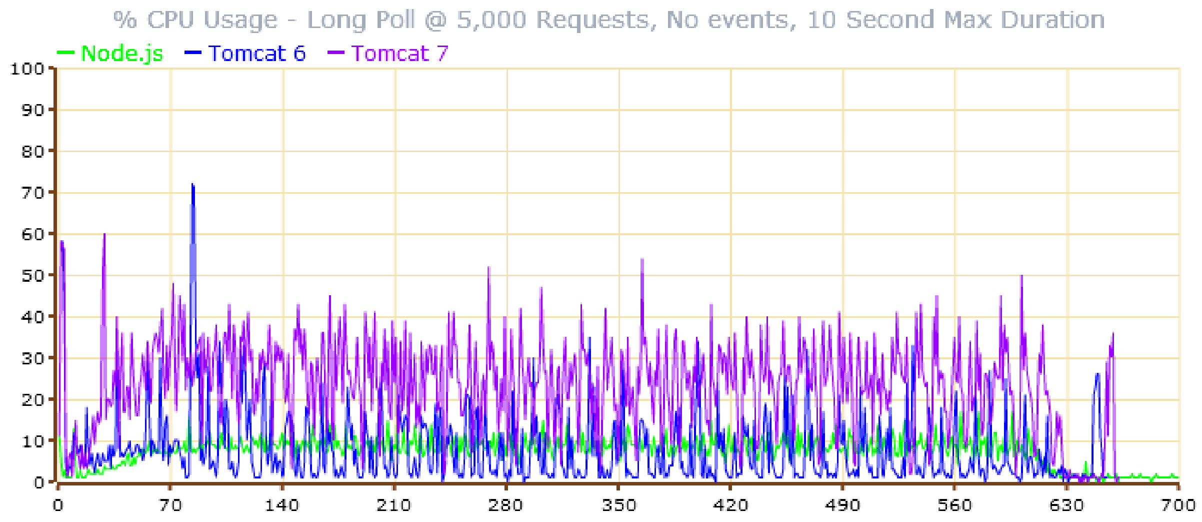
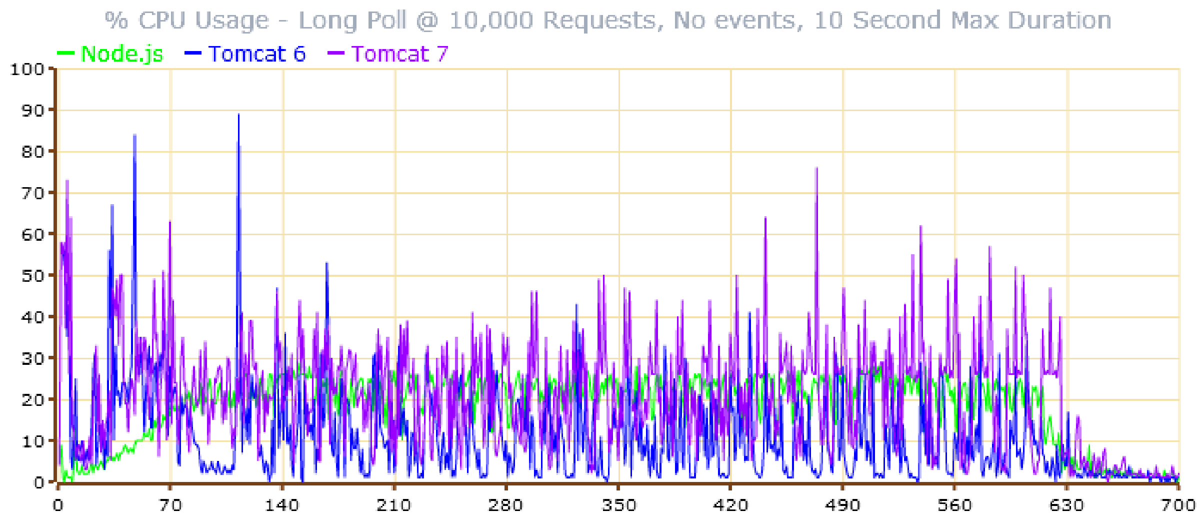
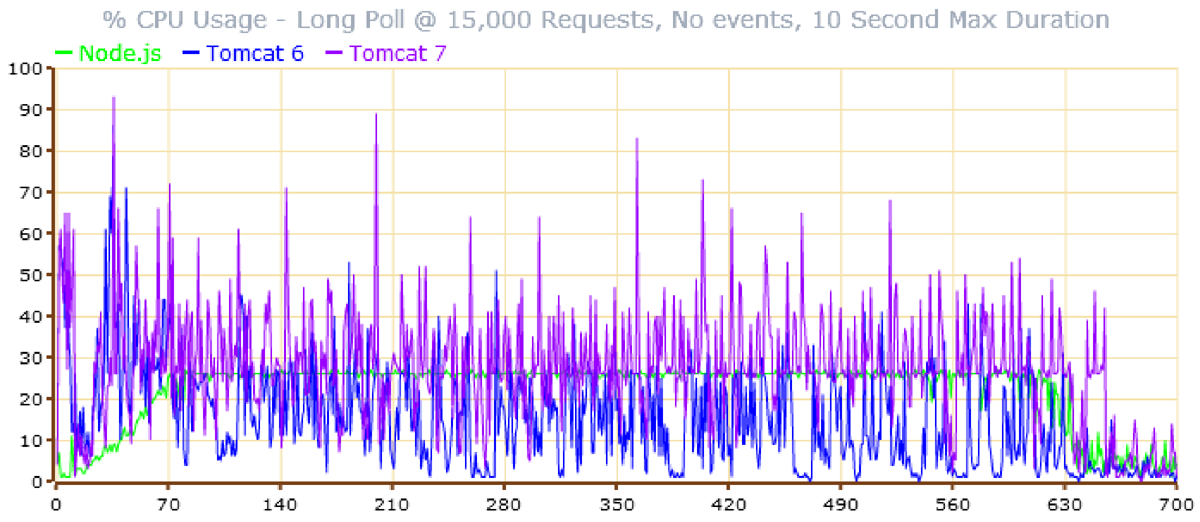

Emit Duration - Comet @ 20,000 Requests, 10 Second Emit Interval
Node.js
Tomcat 6
Tomcat 7
3,100
2,790
2,480
2,170
1,860
1,550
1,240
930
620
310
0
0
67
134
201
268
335
402
469
536
603
670

Figure 10: Long Poll – No Events – 10 Second Request Duration – CPU Usage.








% CPU Usage - Long Poll @ 15,000 Requests, No events, 10 Second Max Duration
Node.js
Tomcat 6
Tomcat 7
100
90
80
70
60
50
40
30
20
10
0
0
70
140
210
280
350
420
490
560
630
700

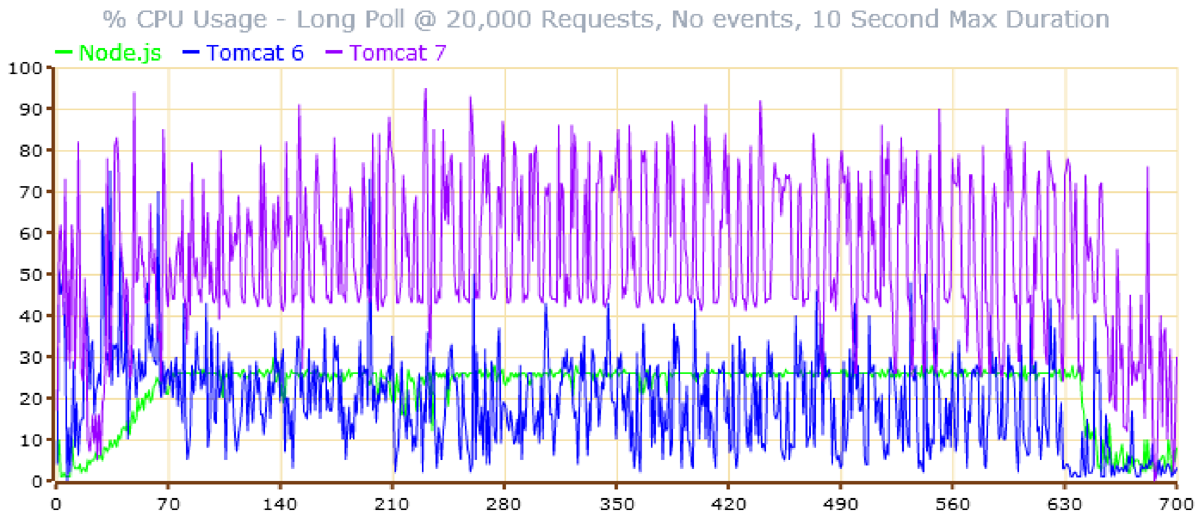
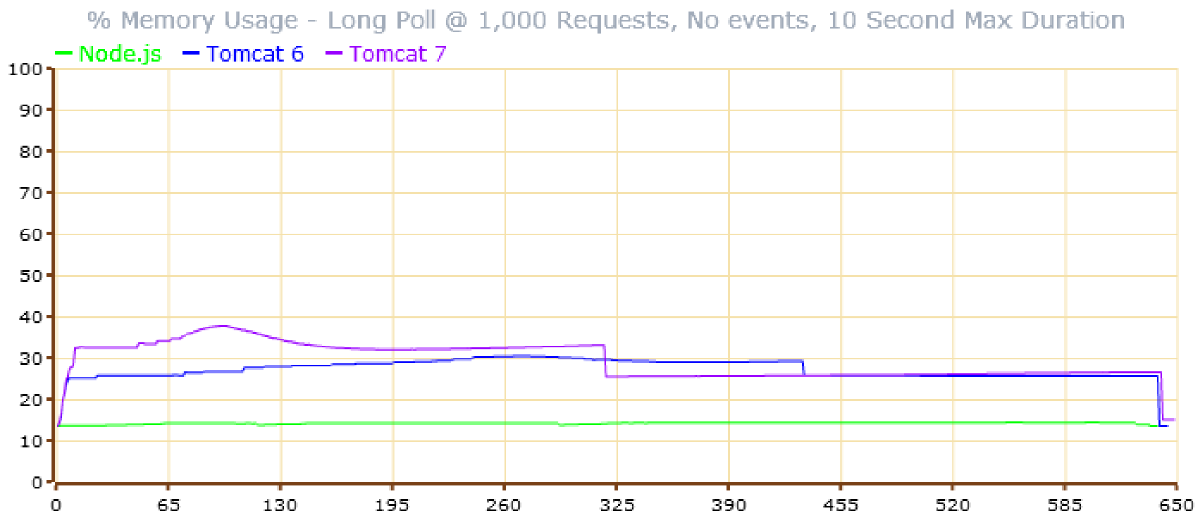
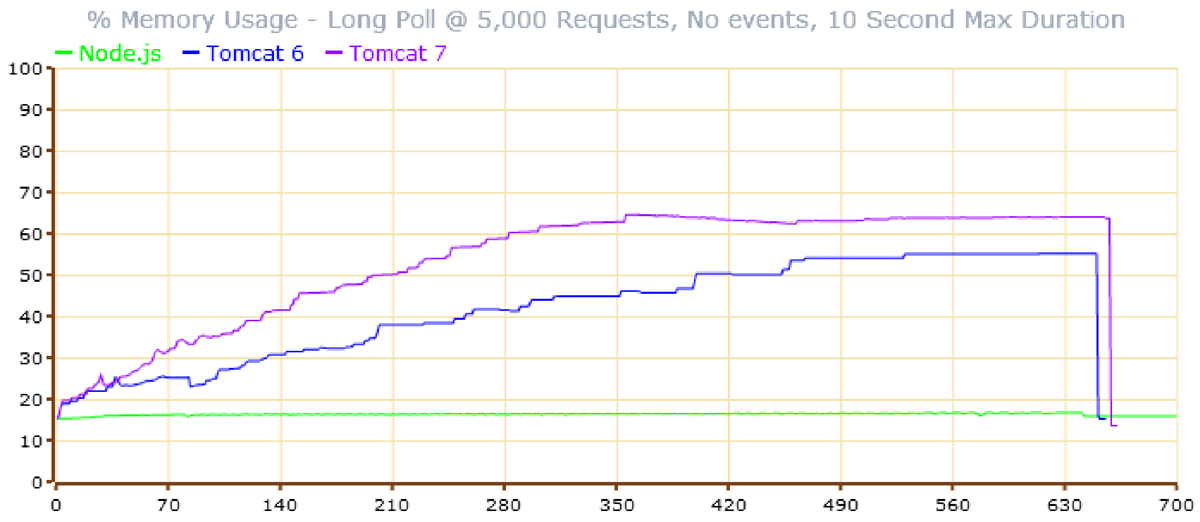
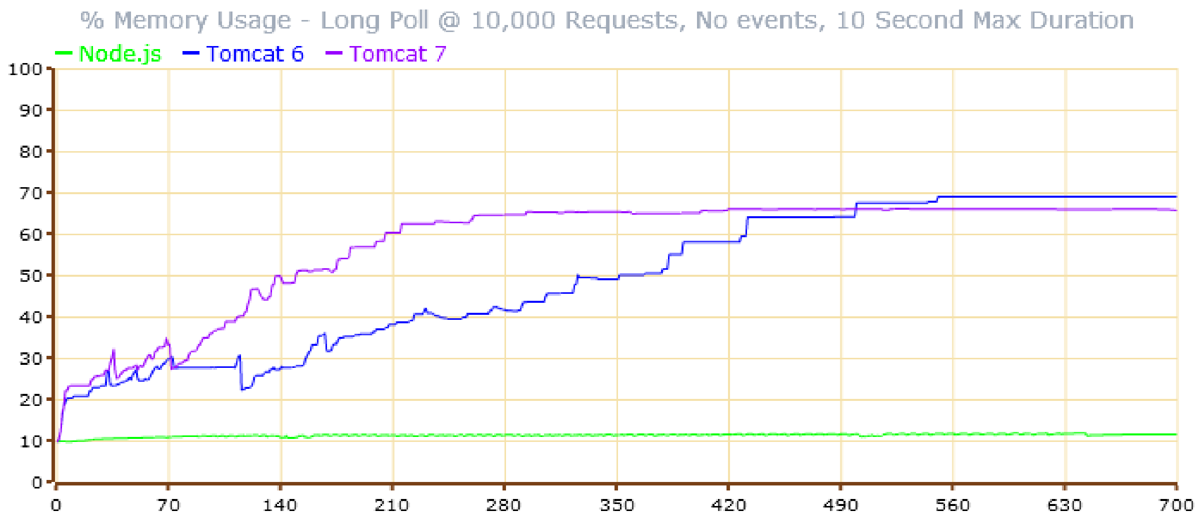
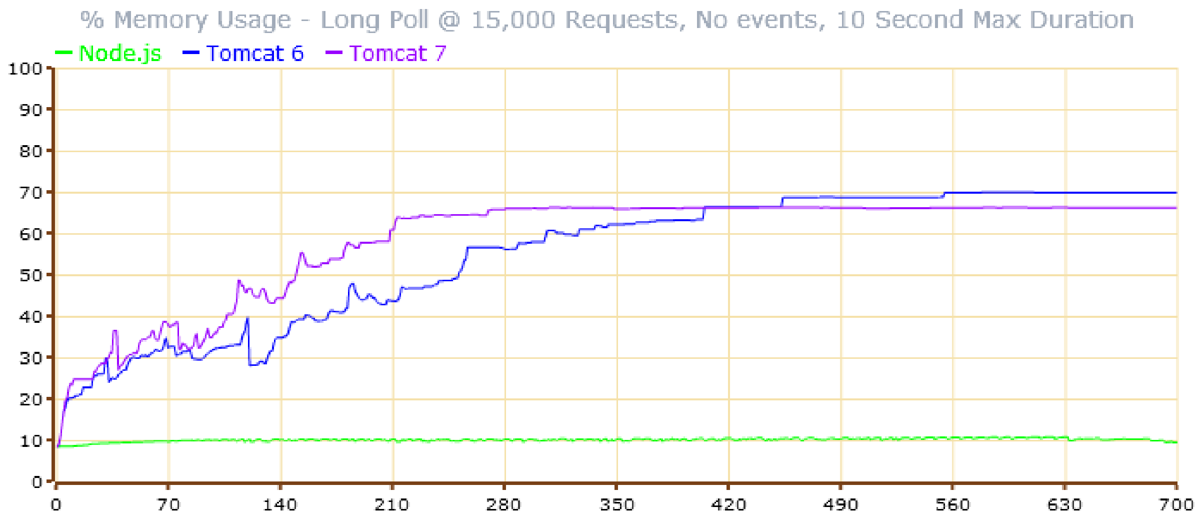

% CPU Usage - Long Poll @ 20,000 Requests, No events, 10 Second Max Duration
Node.js
Tomcat 6
Tomcat 7
100
90
80
70
60
50
40
30
20
10
0
0
70
140
210
280
350
420
490
560
630
700

Figure 11: Long Poll – No Events – 10 Second Request Duration – Memory Usage.








% Memory Usage - Long Poll @ 15,000 Requests, No events, 10 Second Max Duration
Node.js
Tomcat 6
Tomcat 7
0
10
20
30
40
50
60
70
80
90
100
0
70
140
210
280
350
420
490
560
630
700

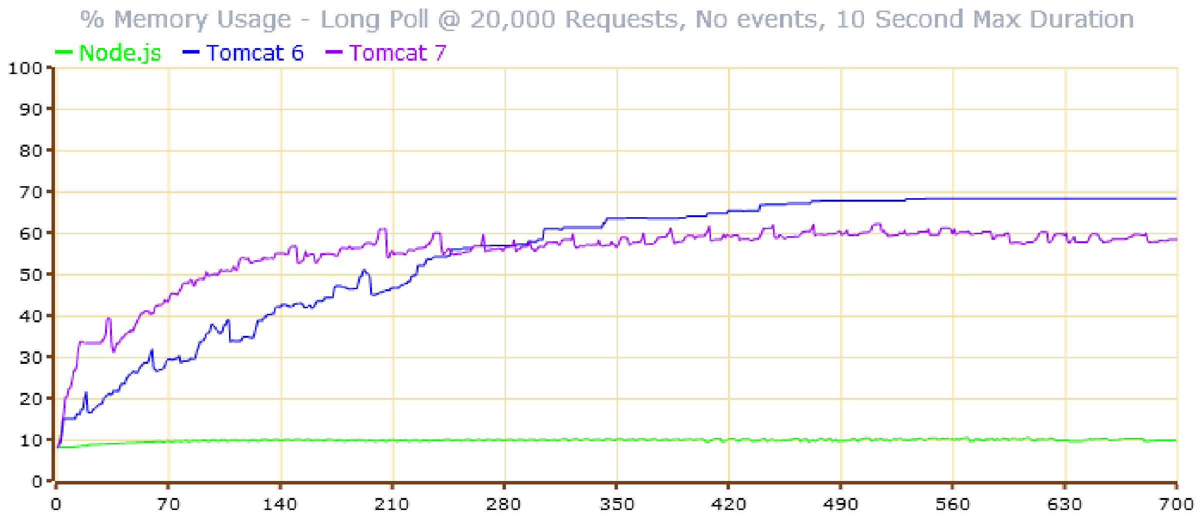
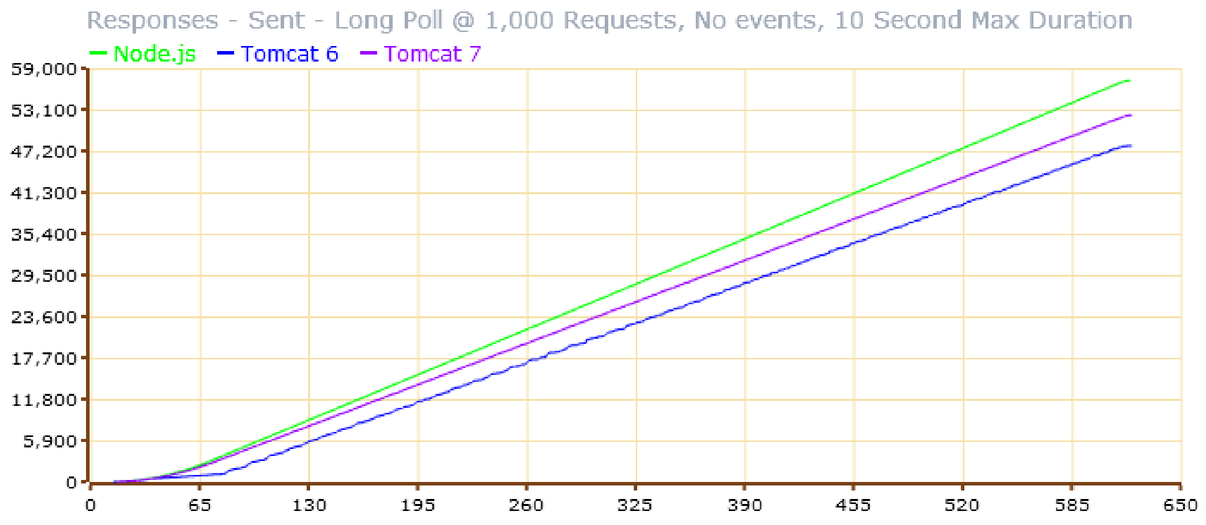
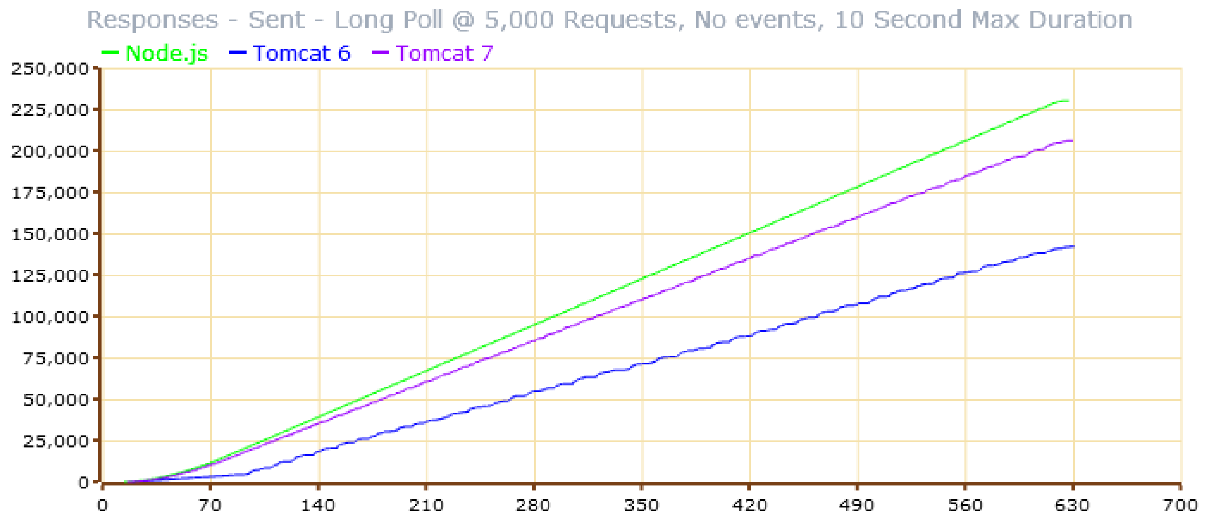
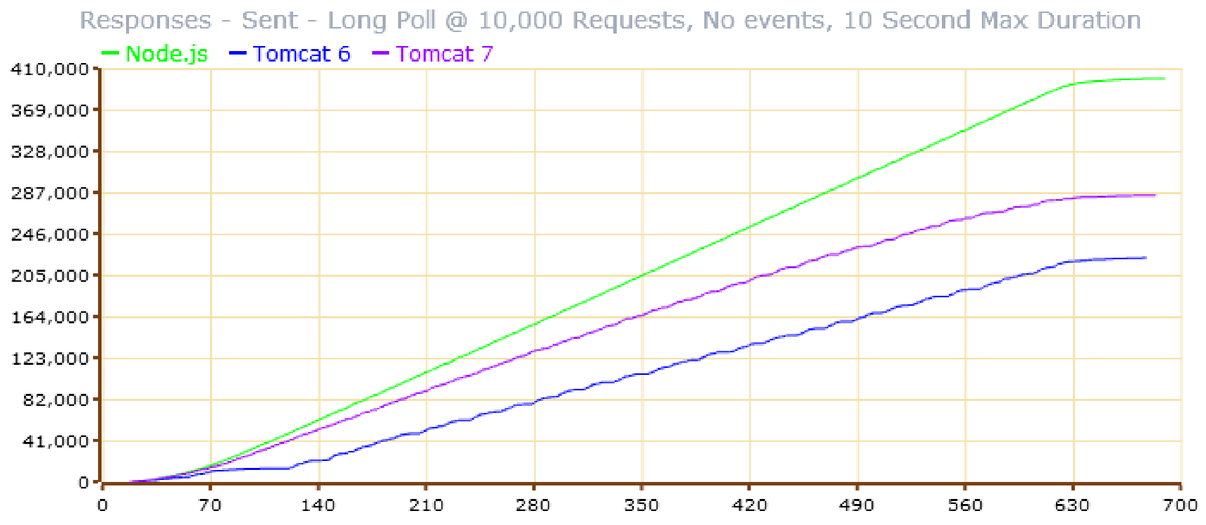

% Memory Usage - Long Poll @ 20,000 Requests, No events, 10 Second Max Duration
Node.js
Tomcat 6
Tomcat 7
0
10
20
30
40
50
60
70
80
90
100
0
70
140
210
280
350
420
490
560
630
700

Figure 12: Long Poll – No Events – 10 Second Request Duration – Responses Sent.







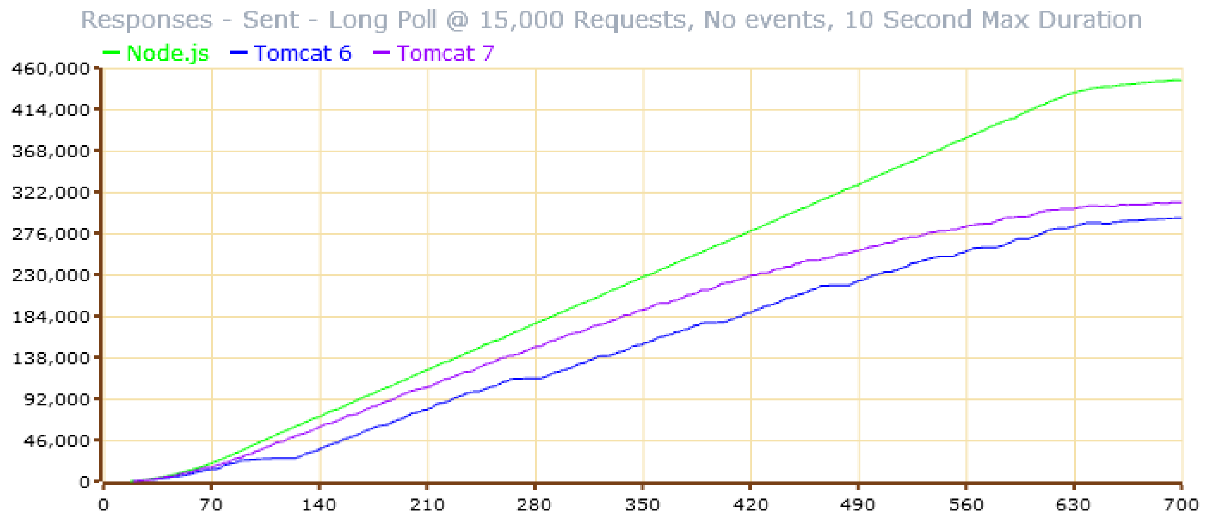
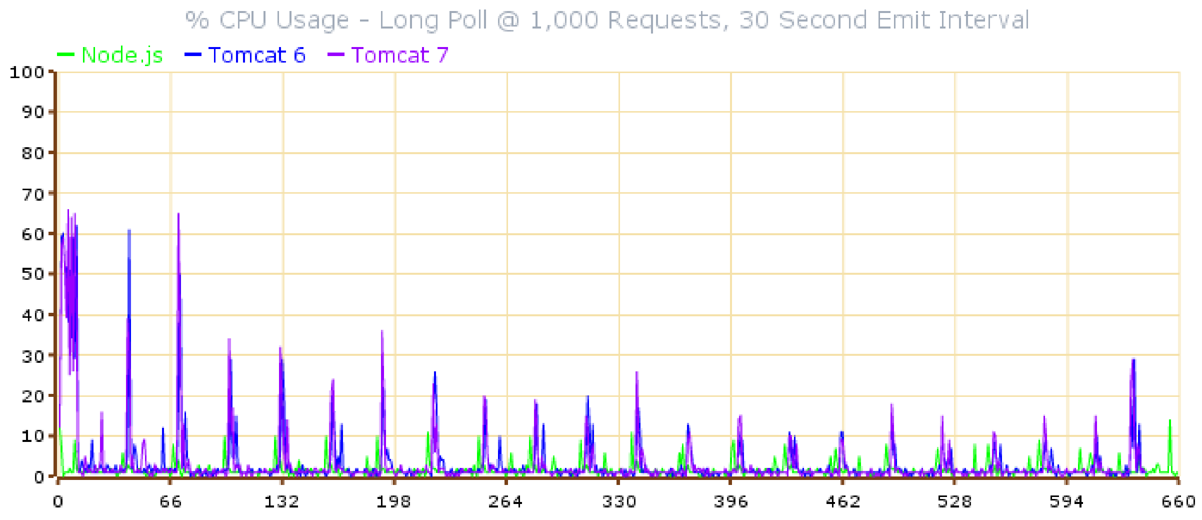
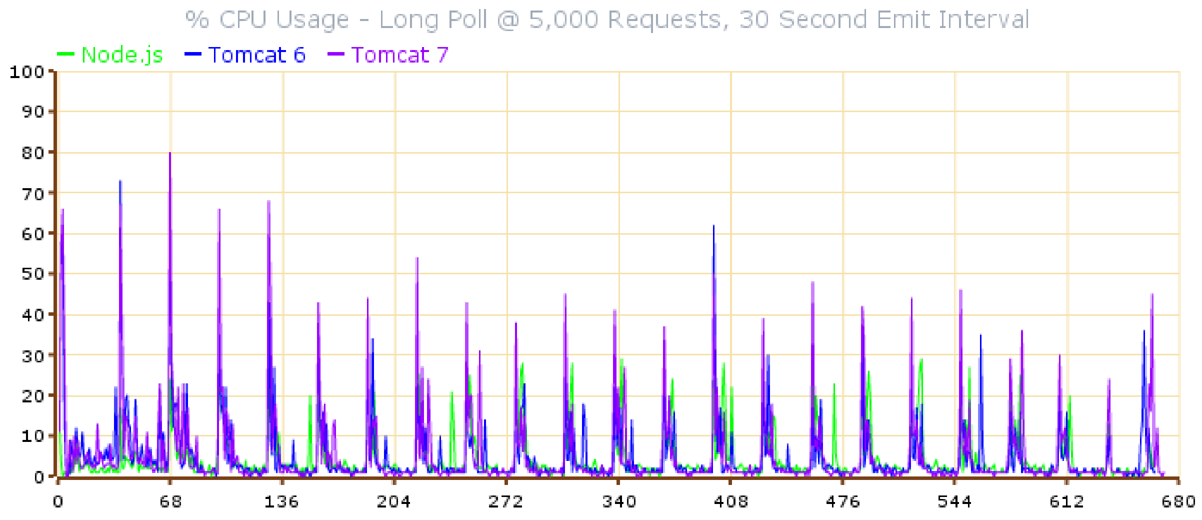
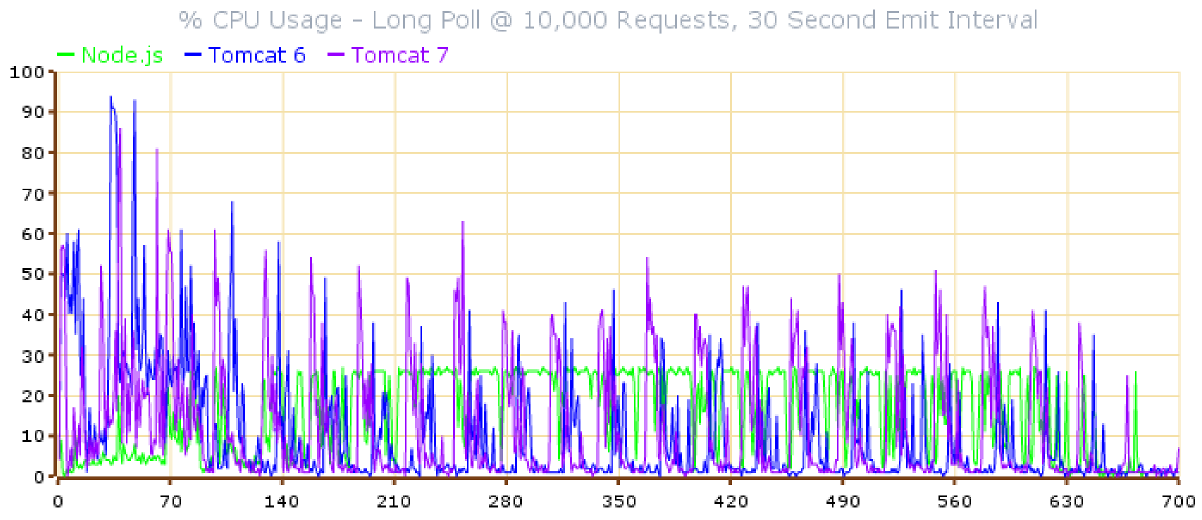
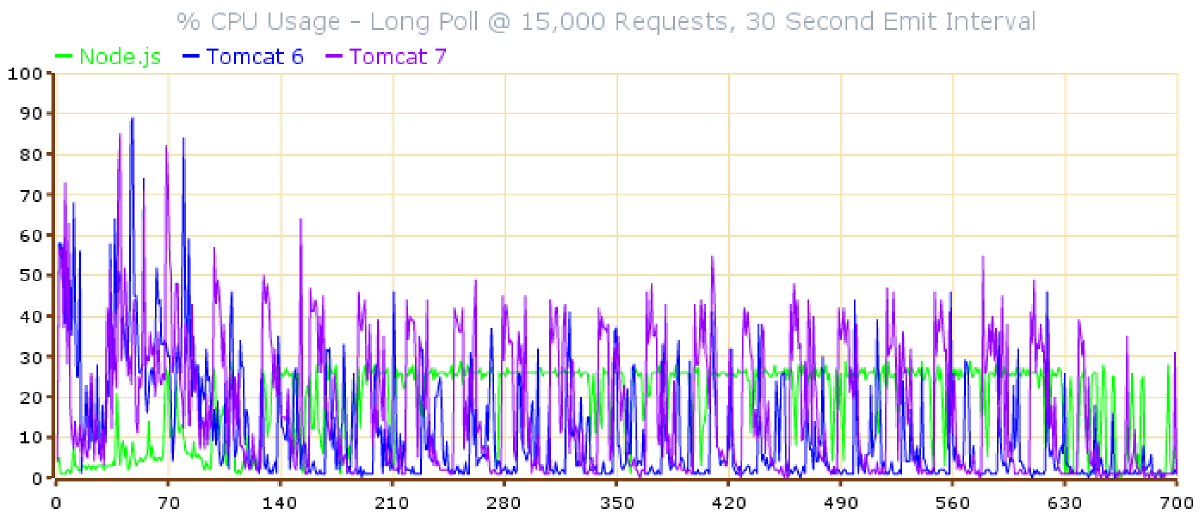

Responses - Sent - Long Poll @ 15,000 Requests, No events, 10 Second Max Duration
Node.js
Tomcat 6
Tomcat 7
460,000
414,000
368,000
322,000
276,000
230,000
184,000
138,000
92,000
46,000
0
0
70
140
210
280
350
420
490
560
630
700

Figure 13: Long Poll – With Events – 30 Second Emit interval – CPU Usage.








% CPU Usage - Long Poll @ 15,000 Requests, 30 Second Emit Interval
Node.js
Tomcat 6
Tomcat 7
100
90
80
70
60
50
40
30
20
10
0
0
70
140
210
280
350
420
490
560
630
700

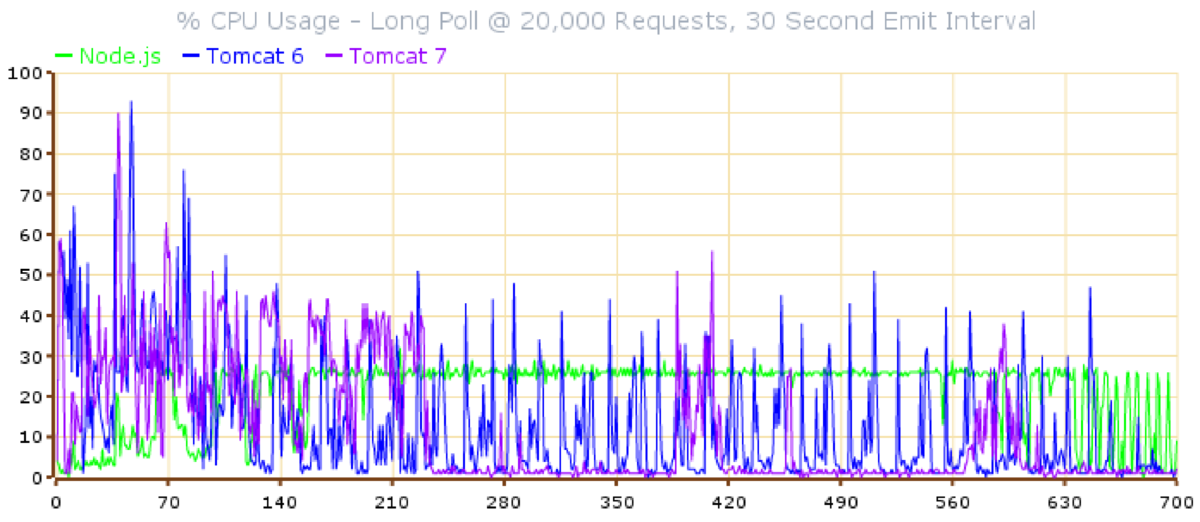
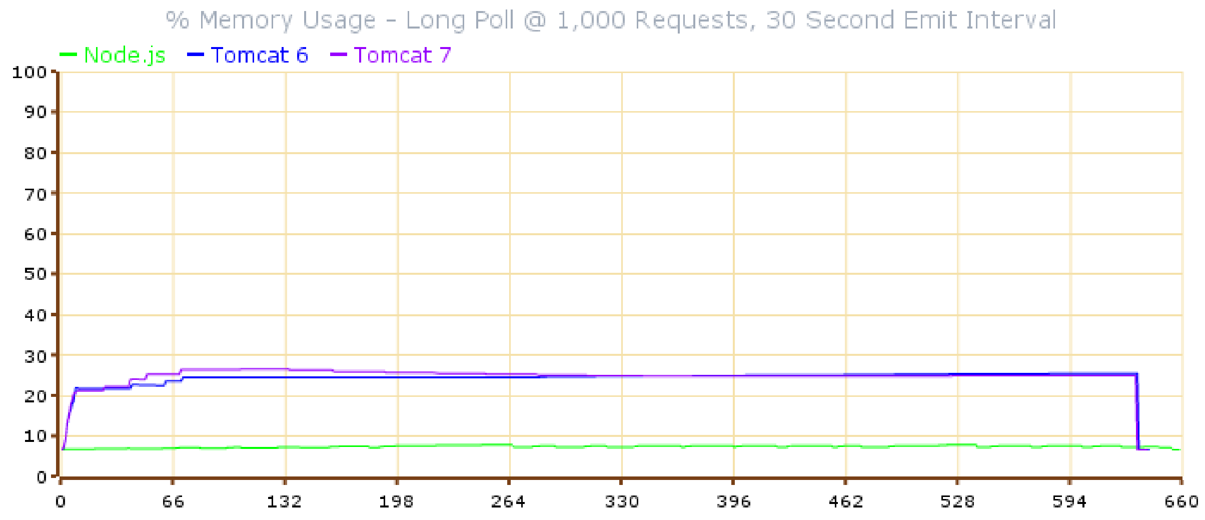
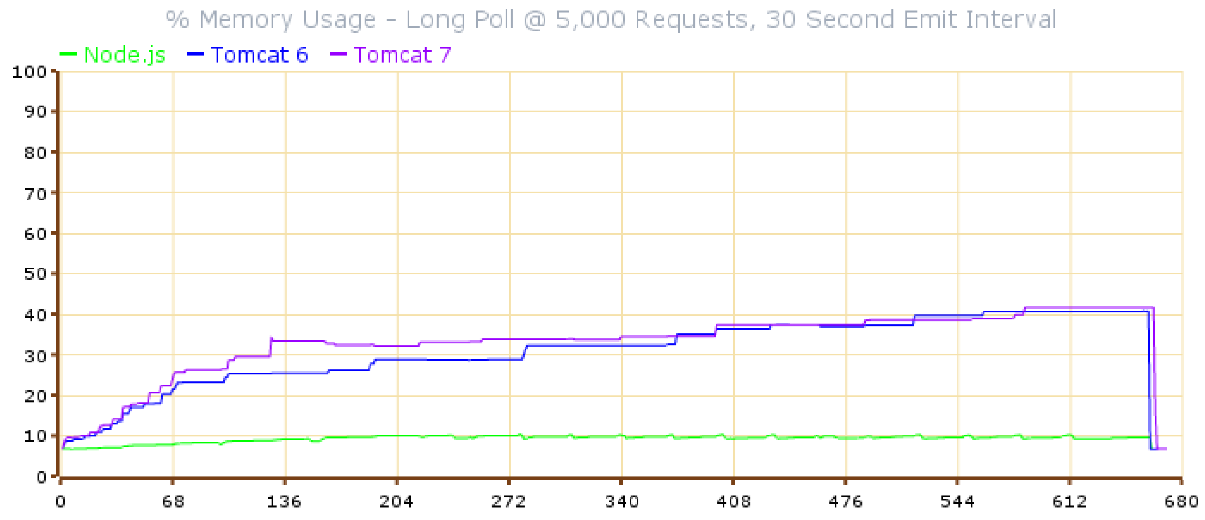
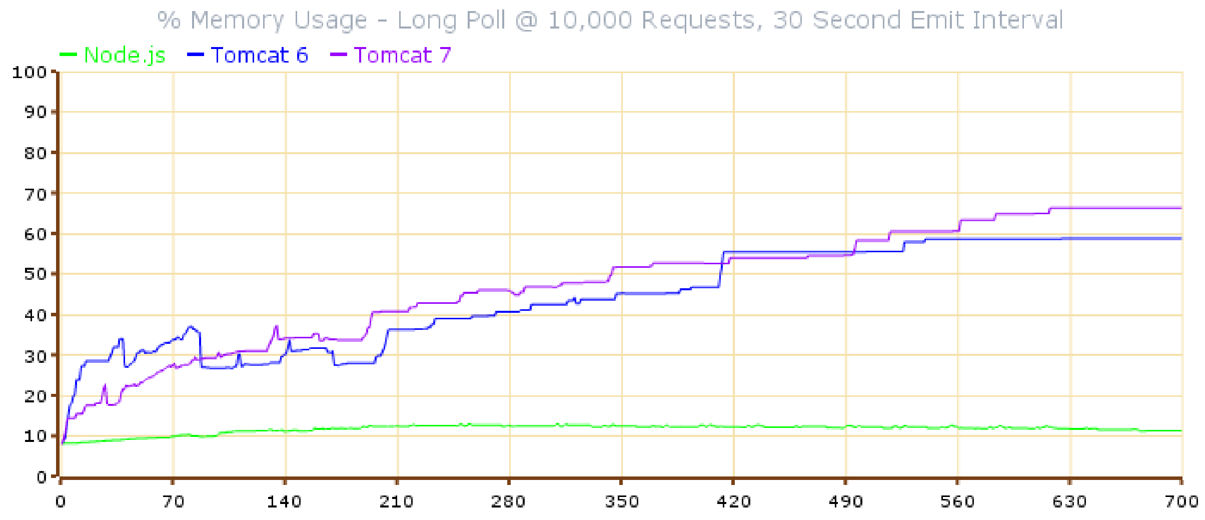
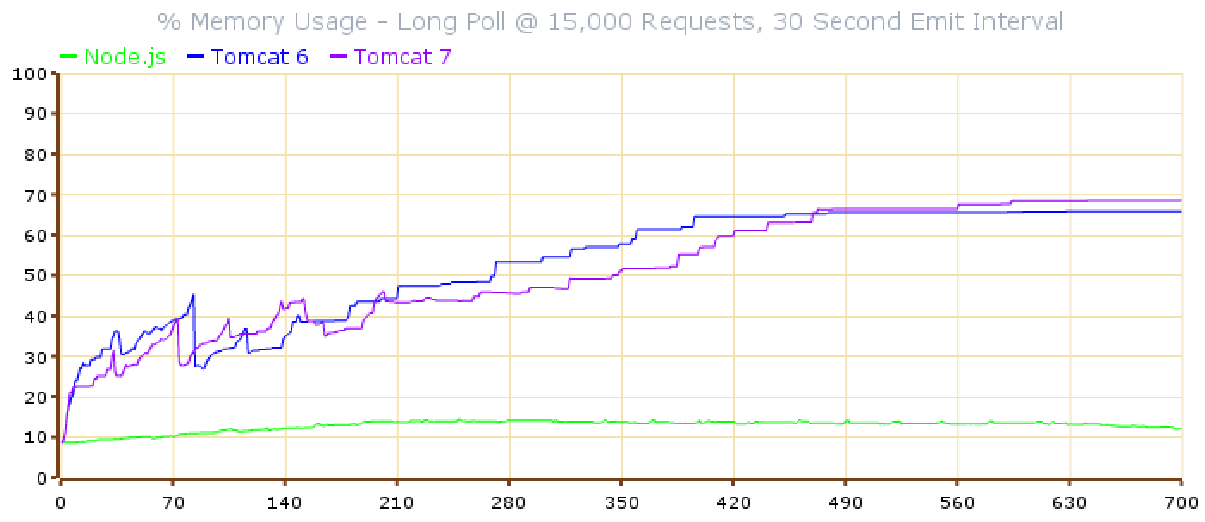

% CPU Usage - Long Poll @ 20,000 Requests, 30 Second Emit Interval
Node.js
Tomcat 6
Tomcat 7
100
90
80
70
60
50
40
30
20
10
0
0
70
140
210
280
350
420
490
560
630
700

Figure 14: Long Poll – With Events – 30 Second Emit interval – Memory Usage.








% Memory Usage - Long Poll @ 15,000 Requests, 30 Second Emit Interval
Node.js
Tomcat 6
Tomcat 7
100
90
80
70
60
50
40
30
20
10
0
0
70
140
210
280
350
420
490
560
630
700

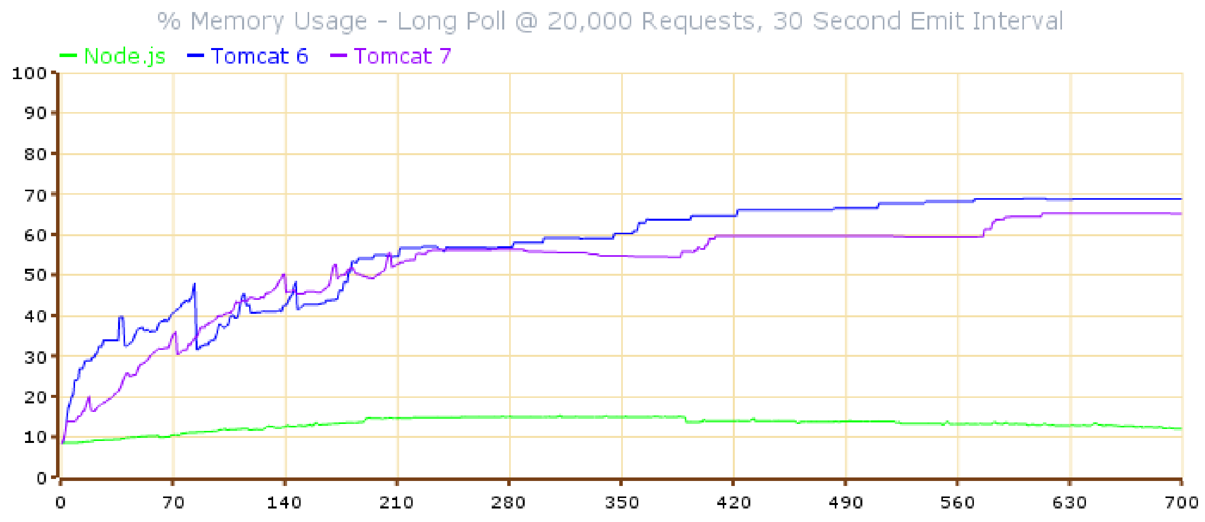
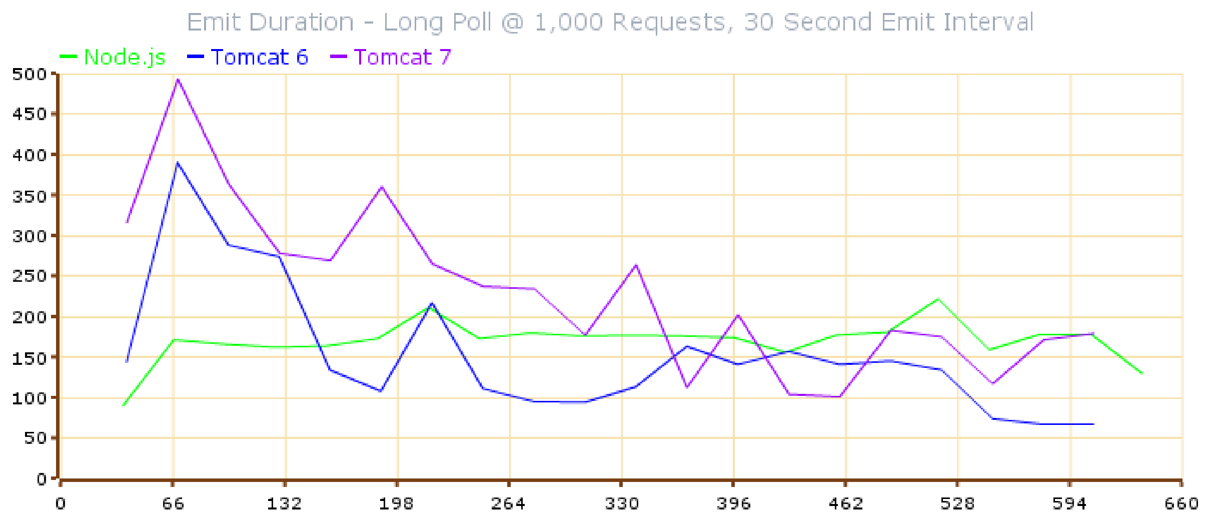
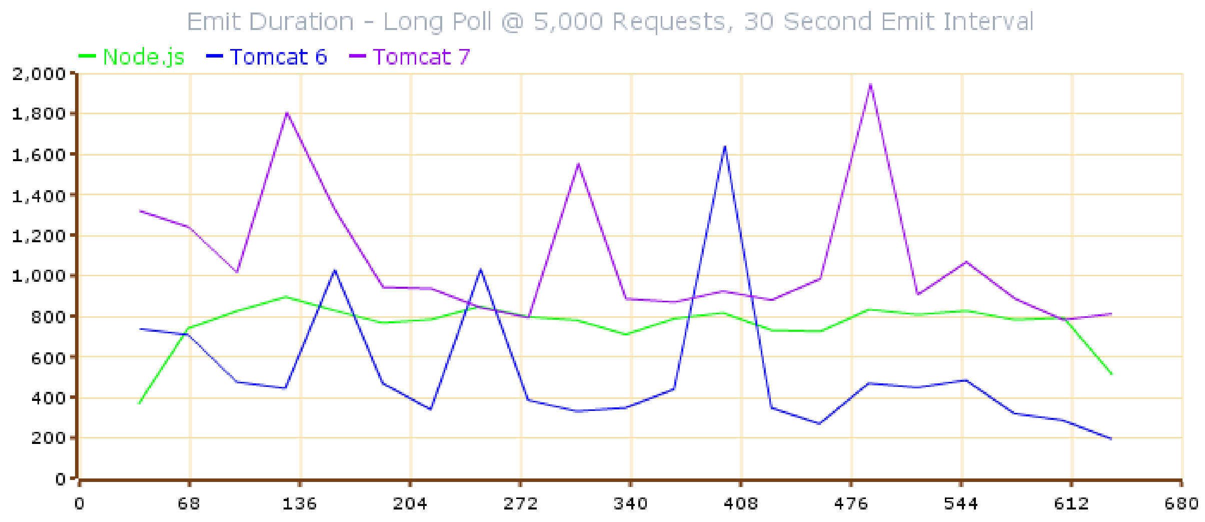
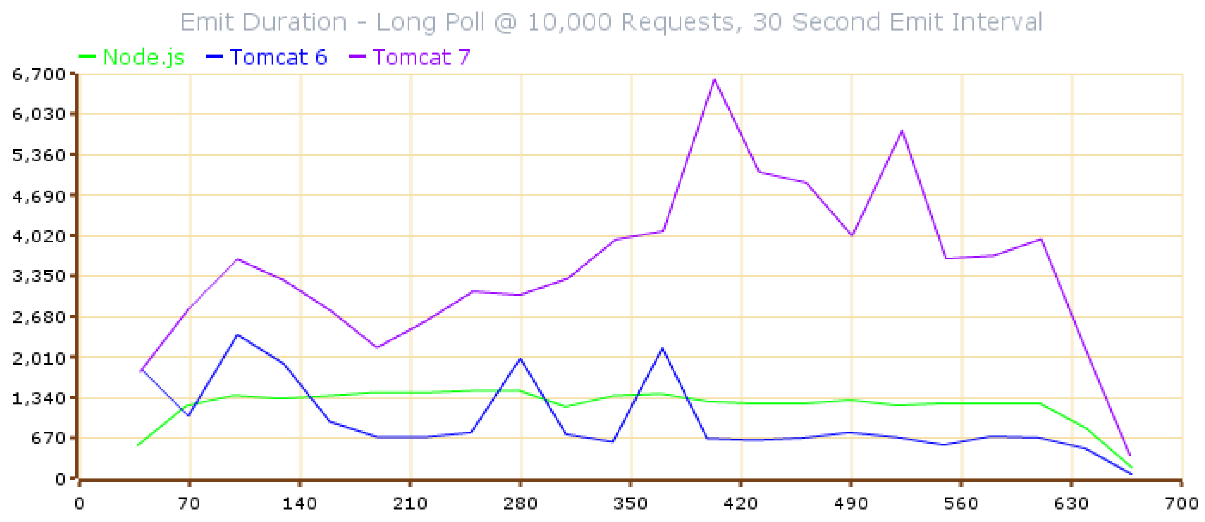
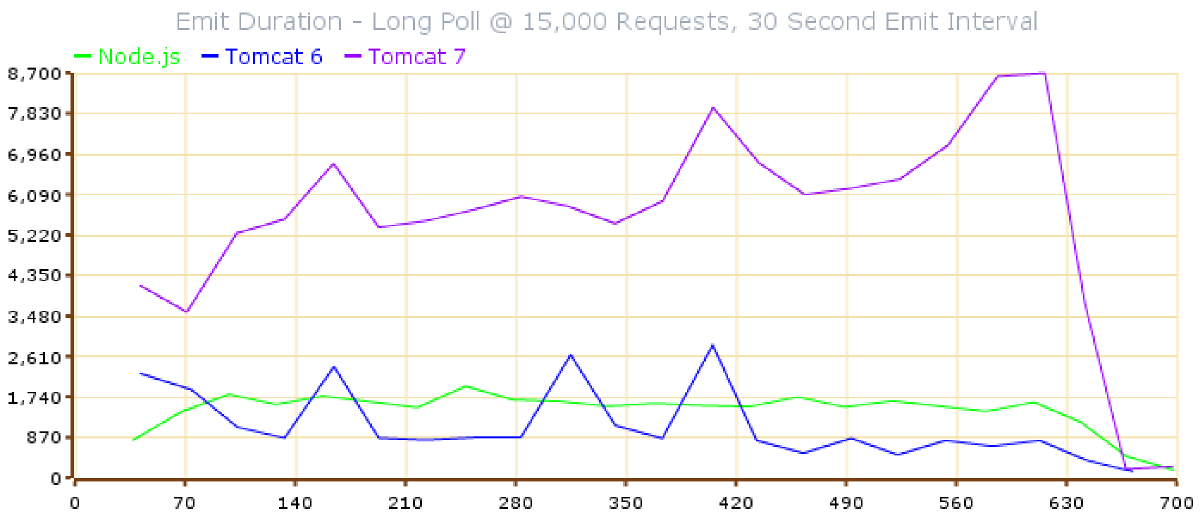

% Memory Usage - Long Poll @ 20,000 Requests, 30 Second Emit Interval
Node.js
Tomcat 6
Tomcat 7
100
90
80
70
60
50
40
30
20
10
0
0
70
140
210
280
350
420
490
560
630
700

Figure 15: Long Poll – With Events – 30 Second Emit interval – Varying Requests.








Emit Duration - Long Poll @ 15,000 Requests, 30 Second Emit Interval
Node.js
Tomcat 6
Tomcat 7
8,700
7,830
6,960
6,090
5,220
4,350
3,480
2,610
1,740
870
0
0
70
140
210
280
350
420
490
560
630
700

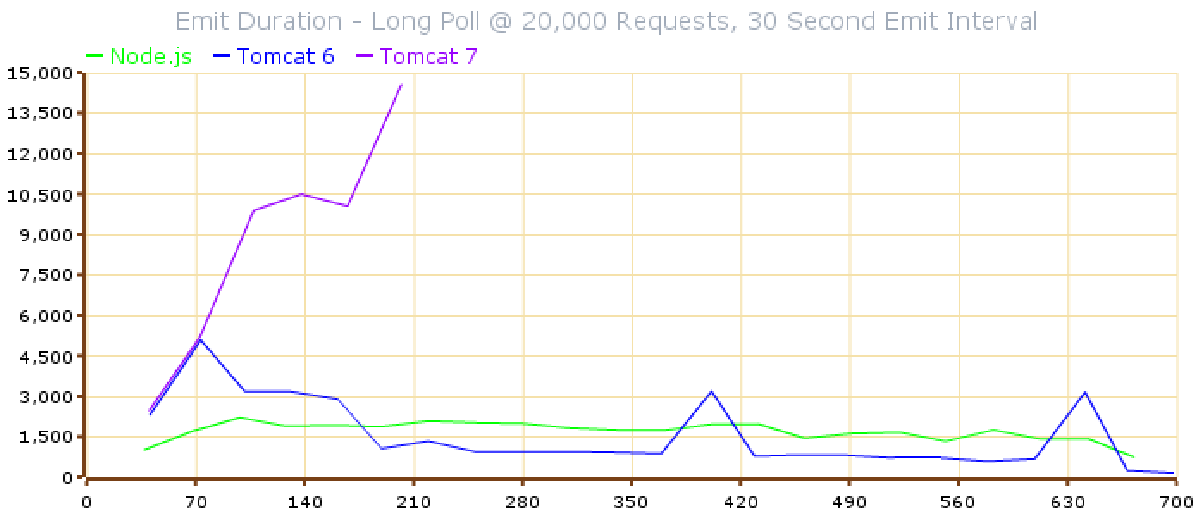
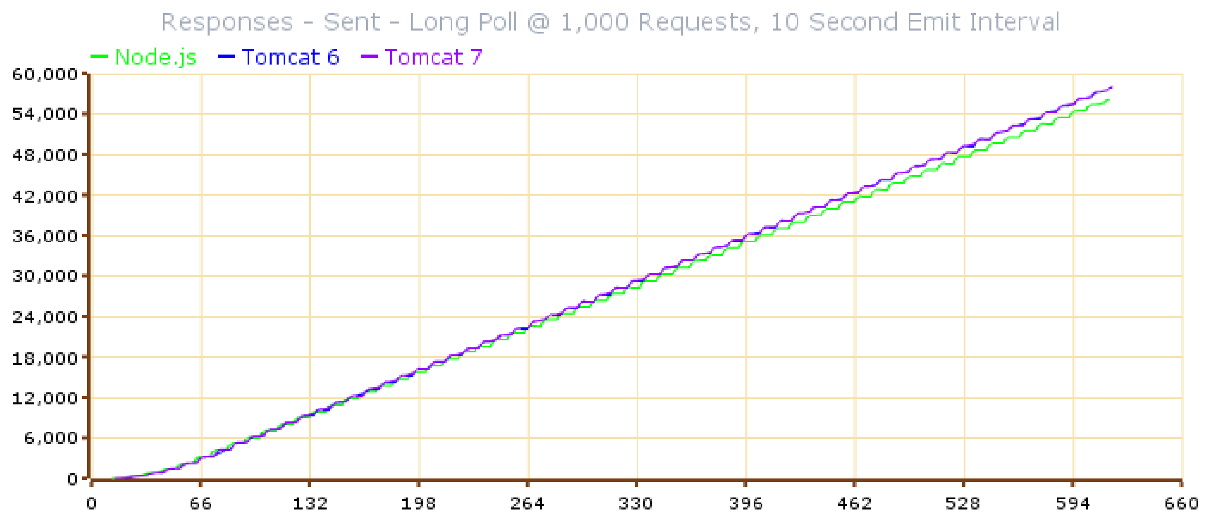
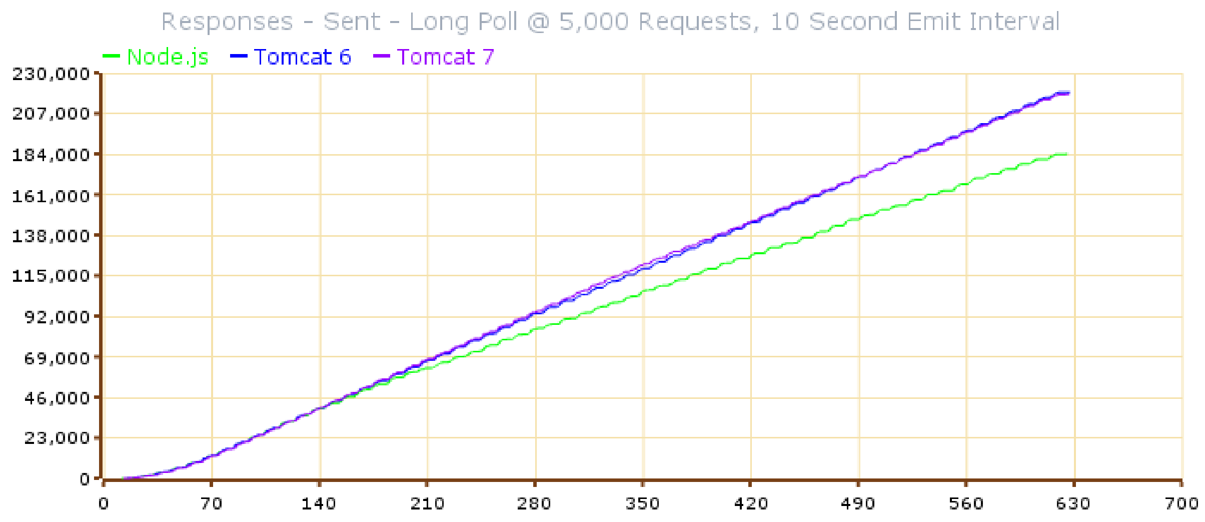
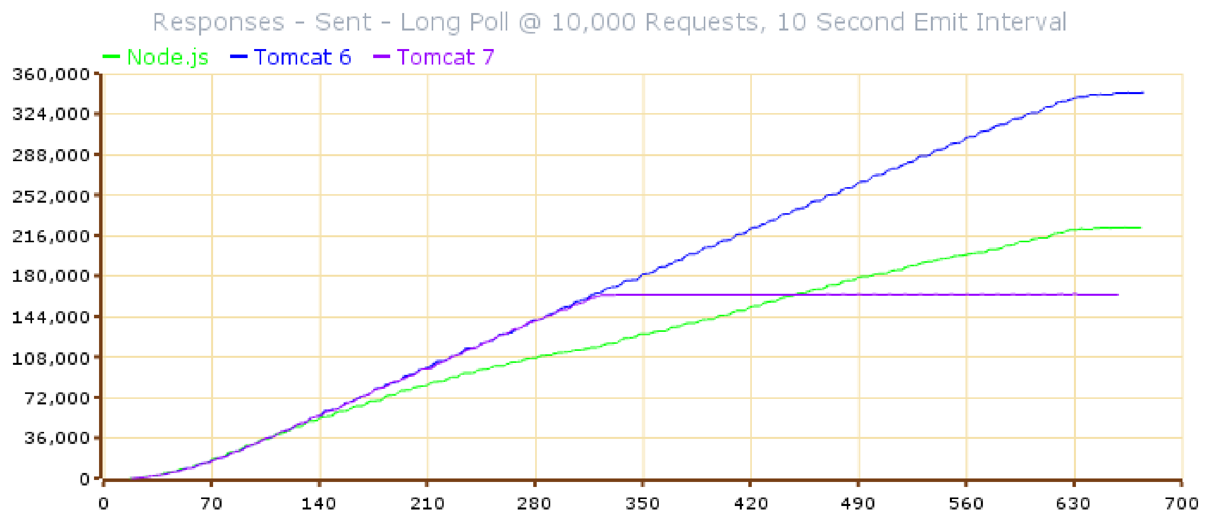
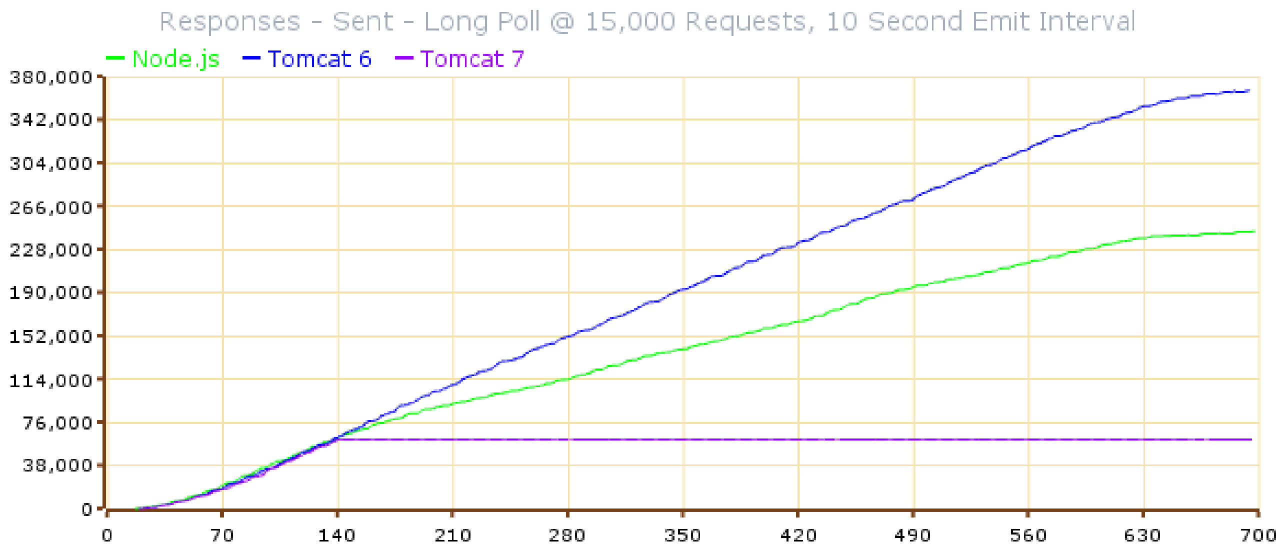

Emit Duration - Long Poll @ 20,000 Requests, 30 Second Emit Interval
Node.js
Tomcat 6
Tomcat 7
15,000
13,500
12,000
10,500
9,000
7,500
6,000
4,500
3,000
1,500
0
0
70
140
210
280
350
420
490
560
630
700

Figure 16: Long Poll – With Events – 10 Second Emit interval – Responses Sent.








Responses - Sent - Long Poll @ 15,000 Requests, 10 Second Emit Interval
Node.js
Tomcat 6
Tomcat 7
380,000
342,000
304,000
266,000
228,000
190,000
152,000
114,000
76,000
38,000
0
0
70
140
210
280
350
420
490
560
630
700

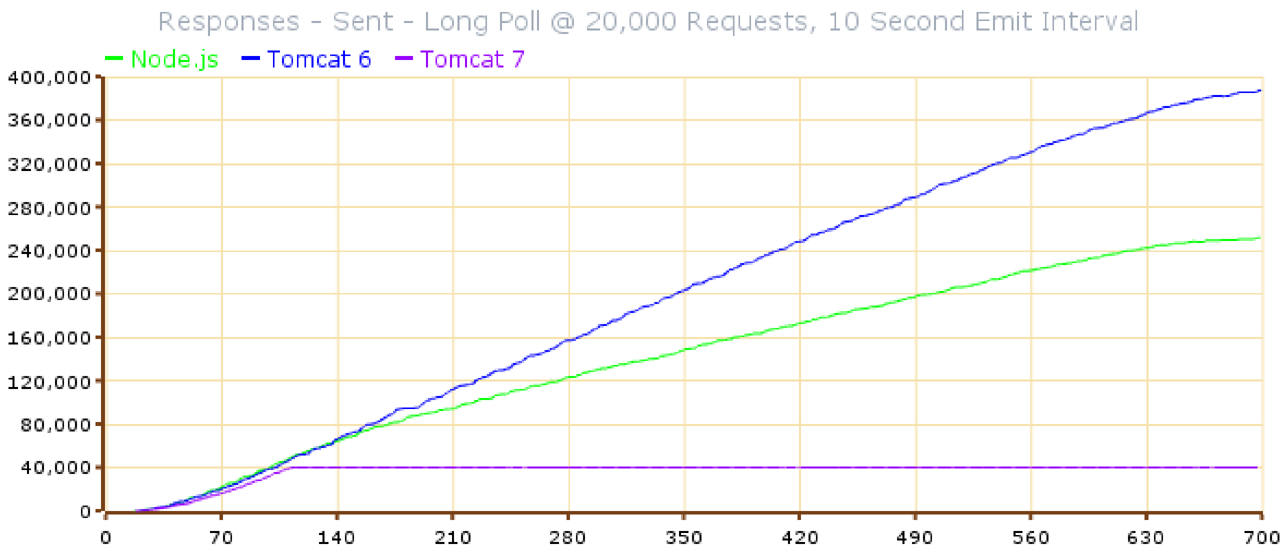
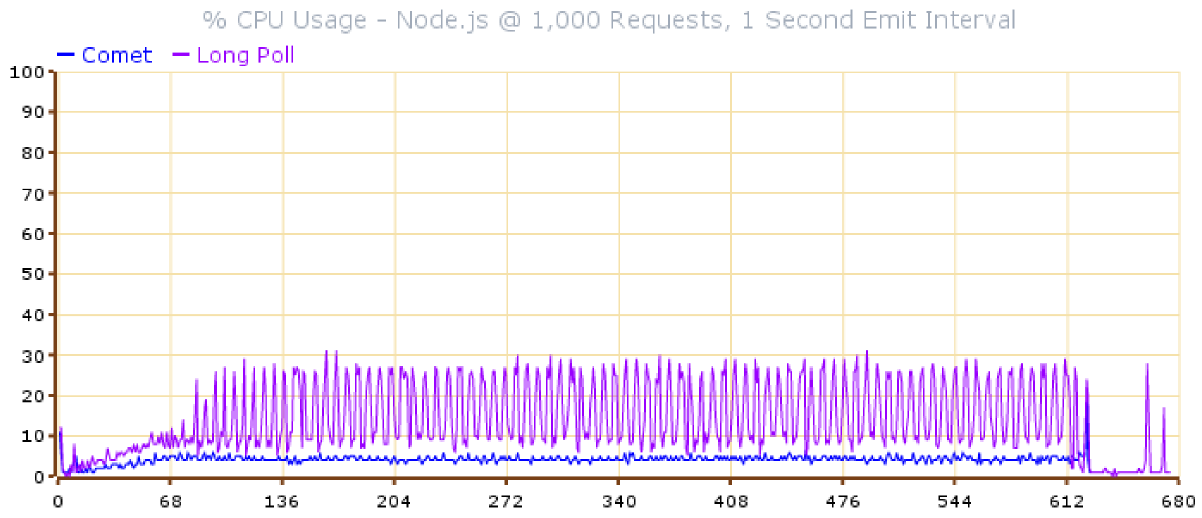
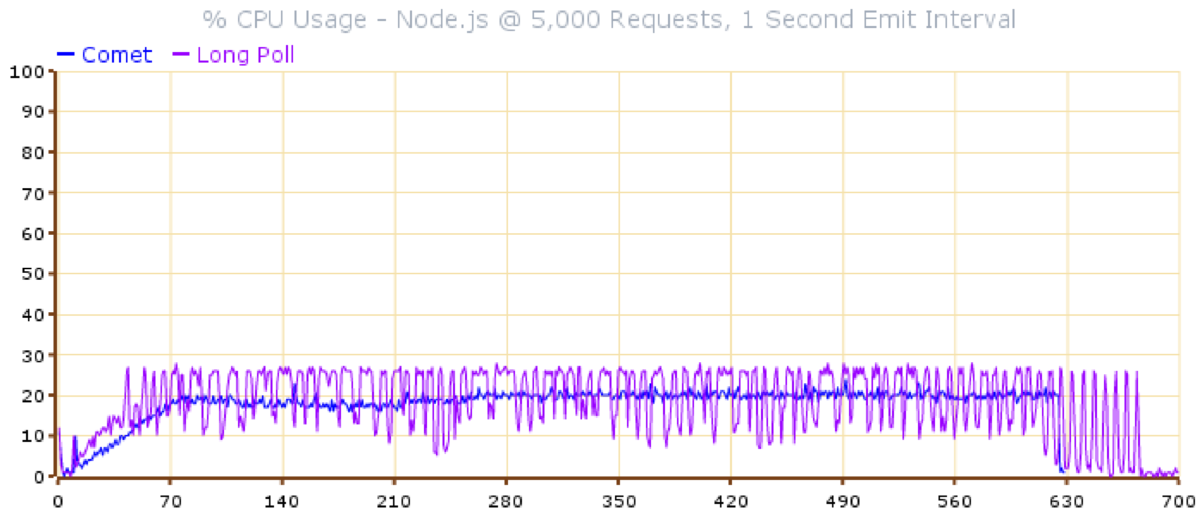
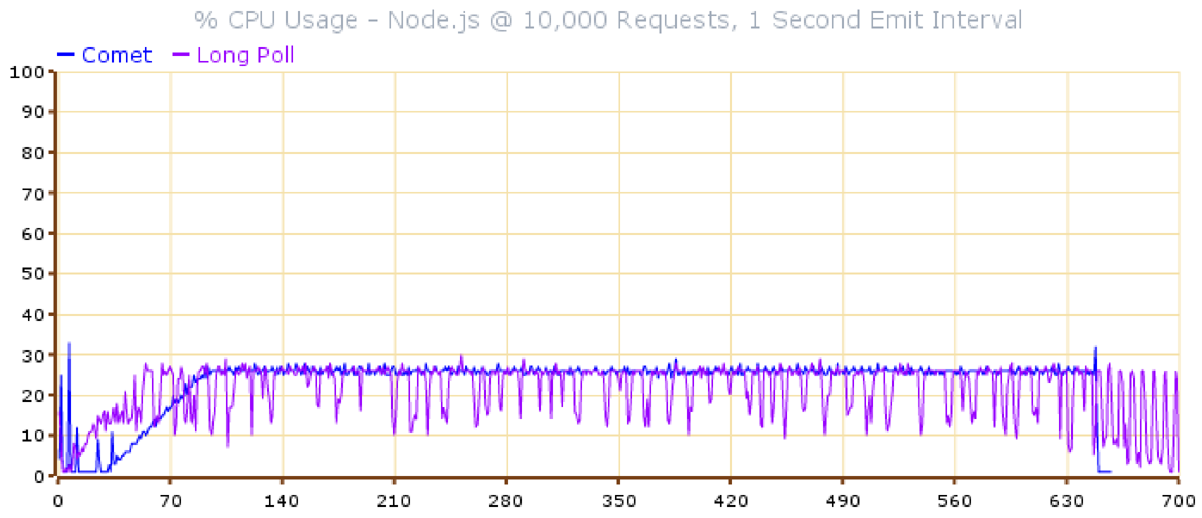
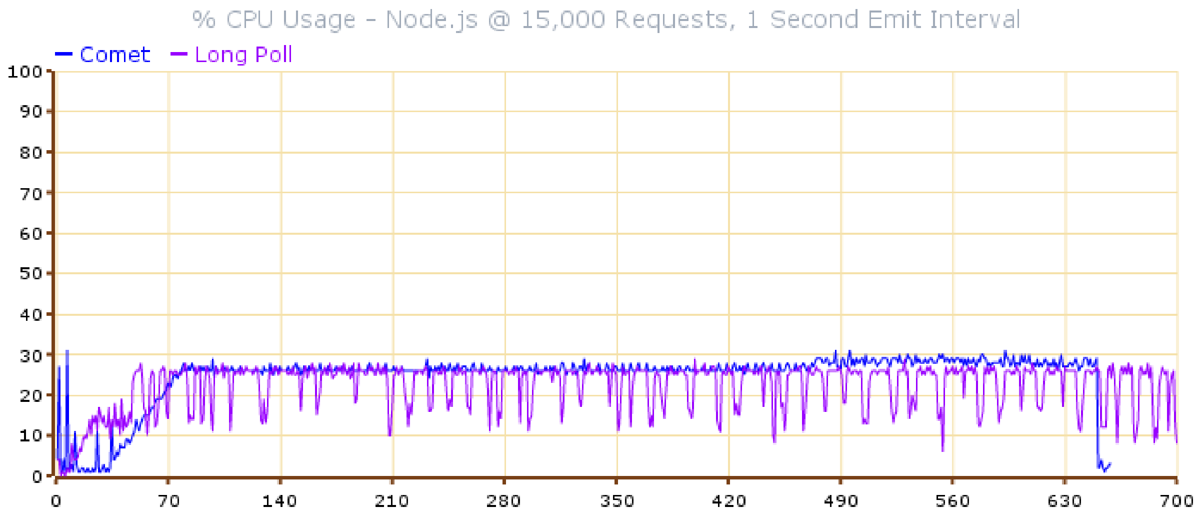

Responses - Sent - Long Poll @ 20,000 Requests, 10 Second Emit Interval
Node.js
Tomcat 6
Tomcat 7
400,000
360,000
320,000
280,000
240,000
200,000
160,000
120,000
80,000
40,000
0
0
70
140
210
280
350
420
490
560
630
700

Figure 17: Node.js – Comet vs Long Poll – 1 Second Emit interval – CPU Usage.

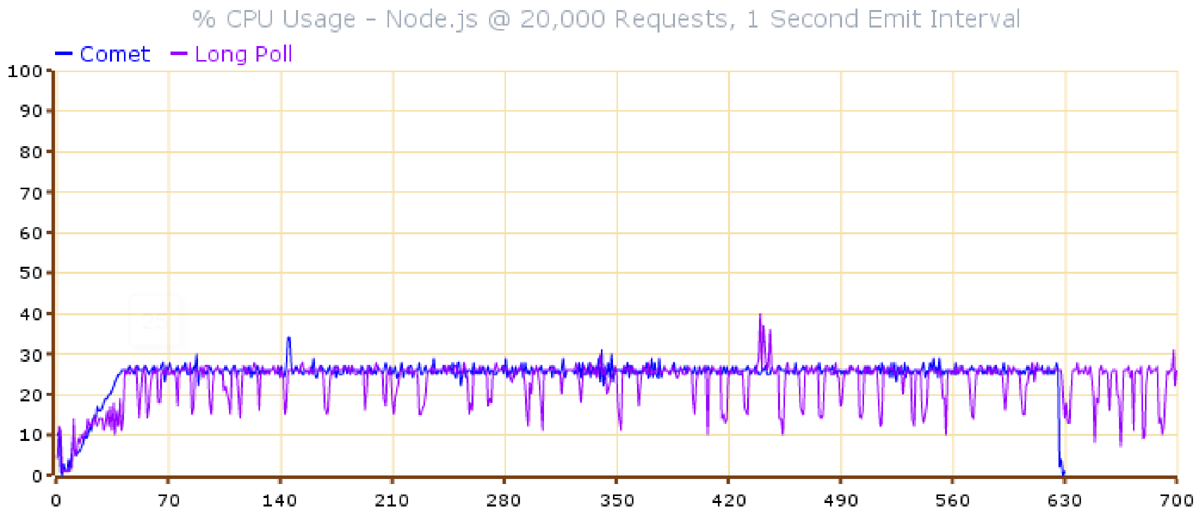








% CPU Usage - Node.js @ 15,000 Requests, 1 Second Emit Interval
Comet
Long Poll
100
90
80
70
60
50
40
30
20
10
0
0
70
140
210
280
350
420
490
560
630
700



% CPU Usage - Node.js @ 20,000 Requests, 1 Second Emit Interval
Comet
Long Poll
100
90
80
70
60
50
40
30
20
10
0
0
70
140
210
280
350
420
490
560
630
700

Figure 18: Node.js – Comet vs Long Poll – 30 Second Emit interval @ 2,0000 connections.

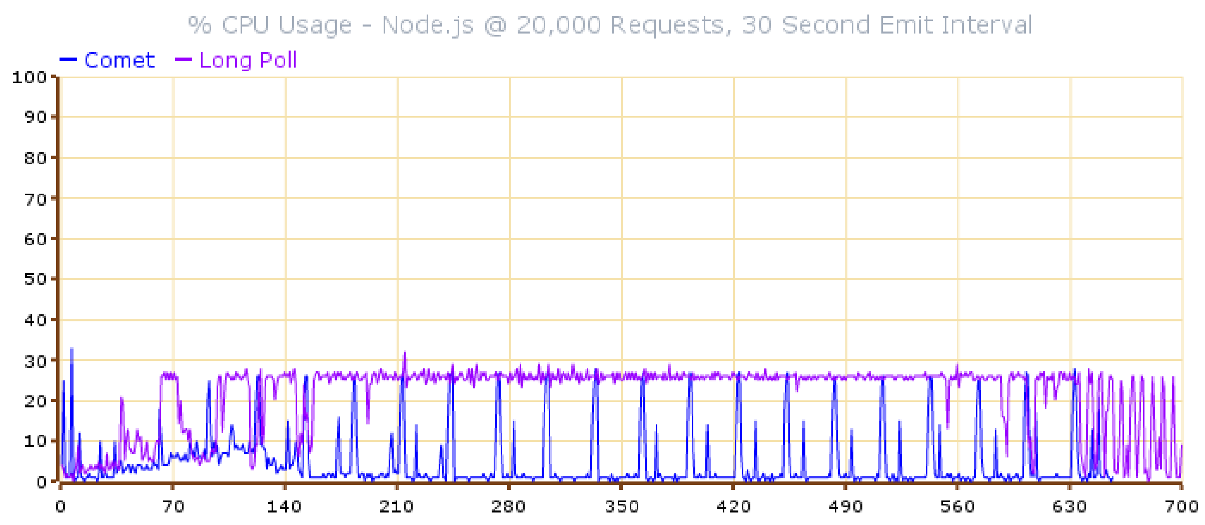
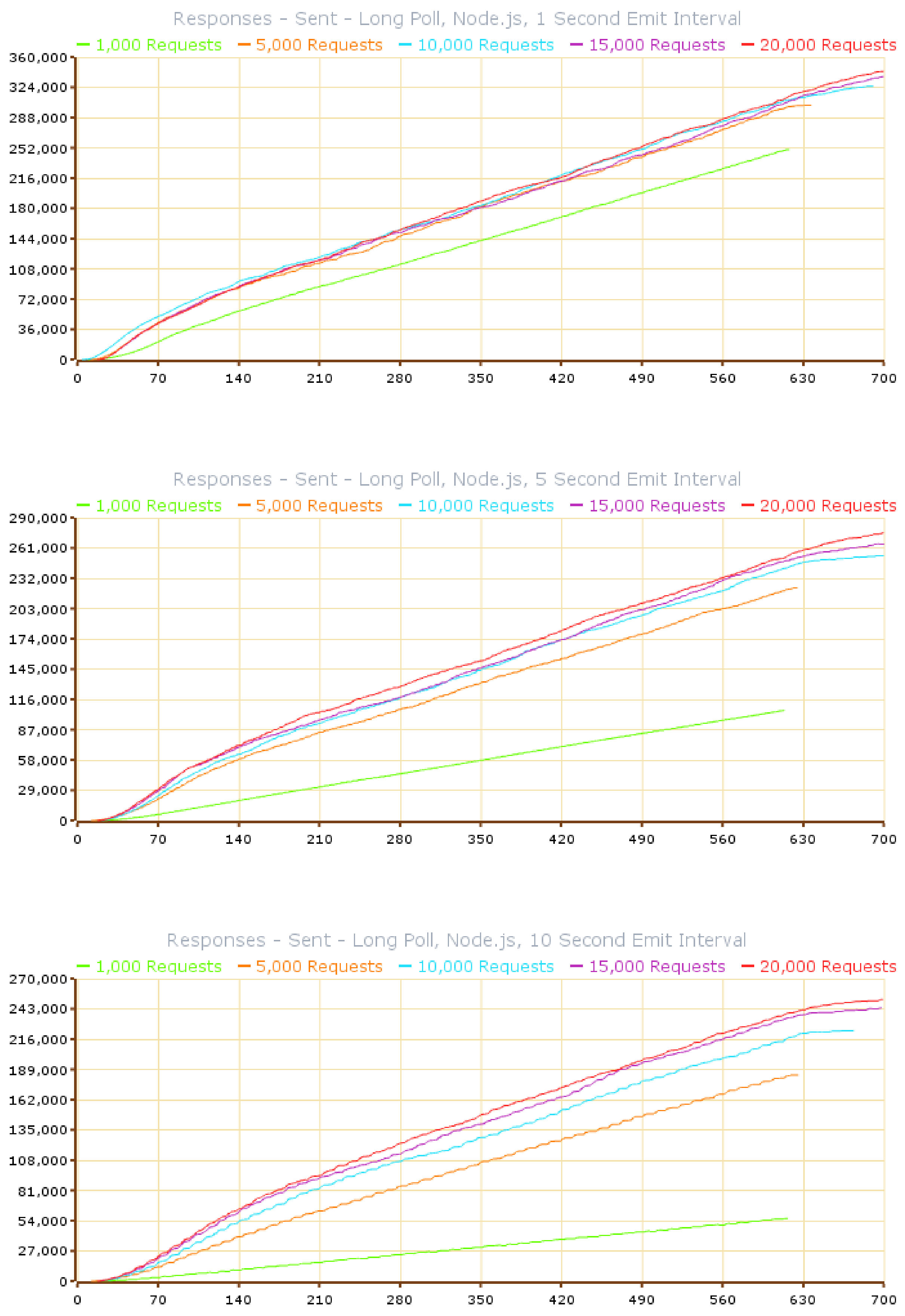


Figure 19: Node.js – Responses Sent – 1 to 30 Second Emit interval.



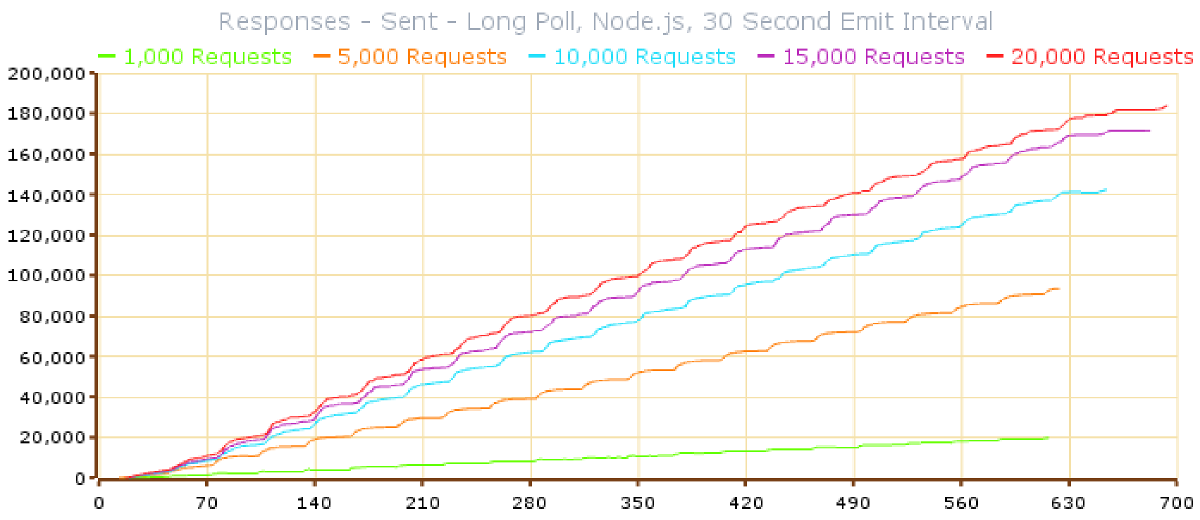
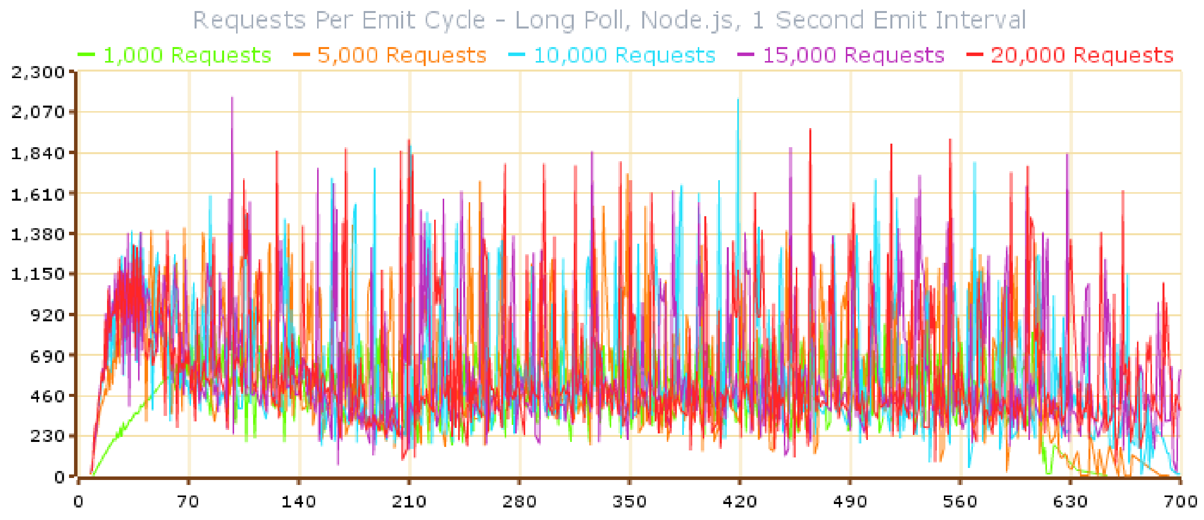

Responses - Sent - Long Poll, Node.js, 30 Second Emit Interval
1,000 Requests
5,000 Requests
10,000 Requests
15,000 Requests
20,000 Requests
200,000
180,000
160,000
140,000
120,000
100,000
80,000
60,000
40,000
20,000
0
0
70
140
210
280
350
420
490
560
630
700

Figure 20: Node.js – Requests per Emit Cycle – 1 Second and 30 Second Emit Interval.



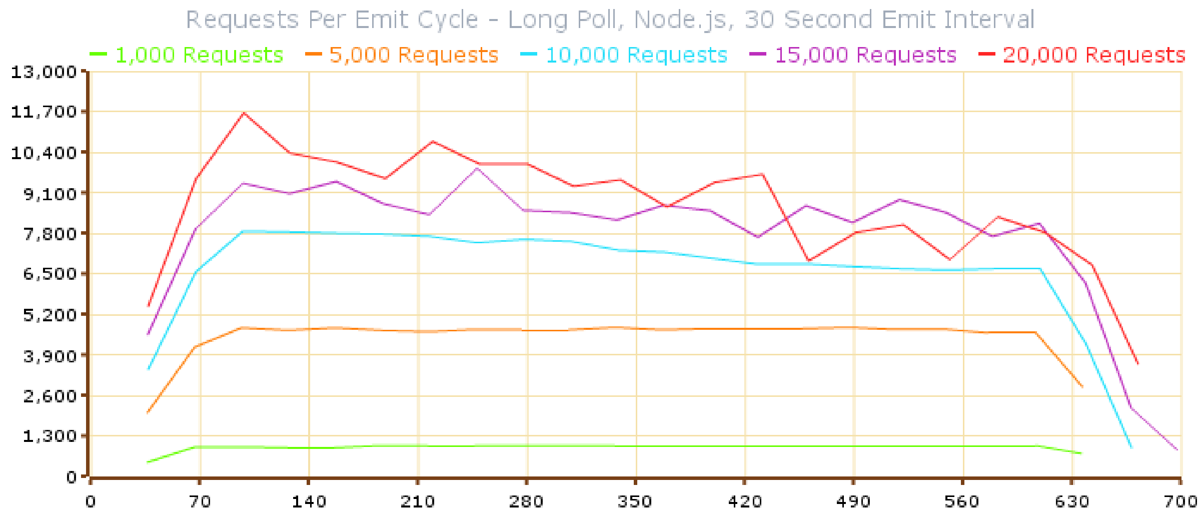
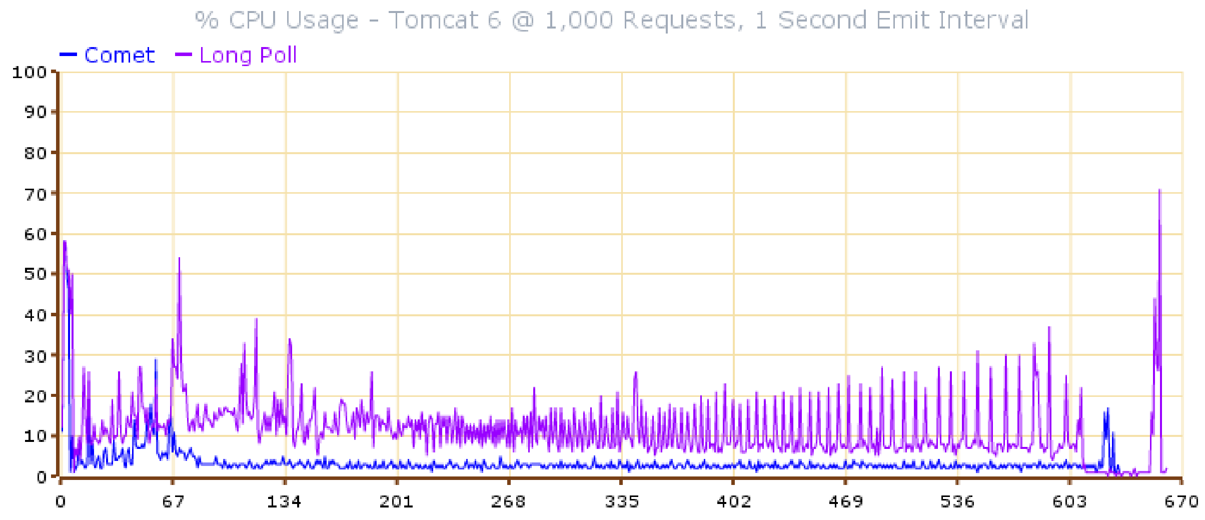
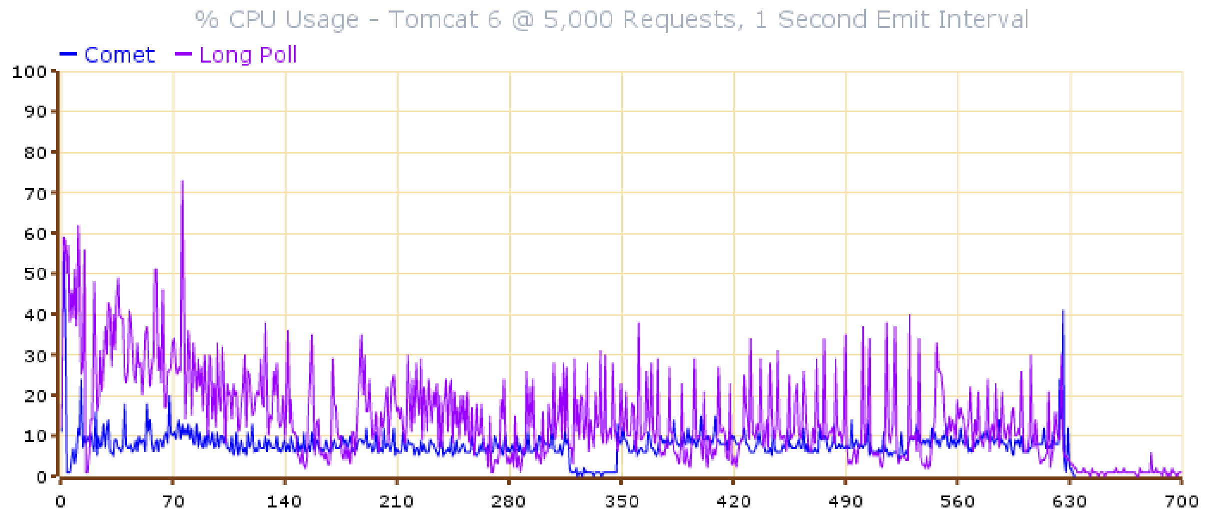
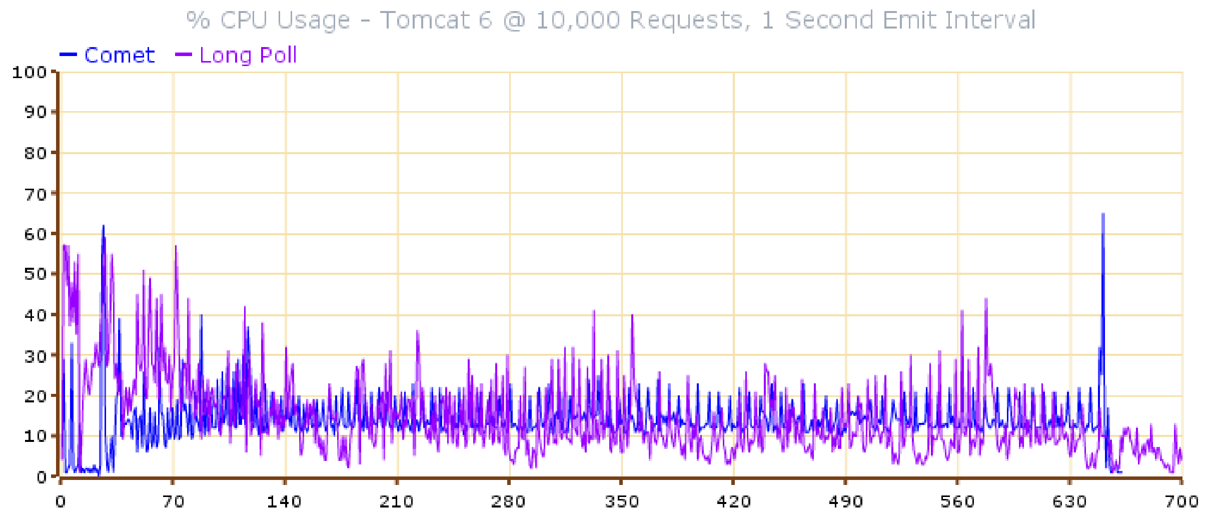
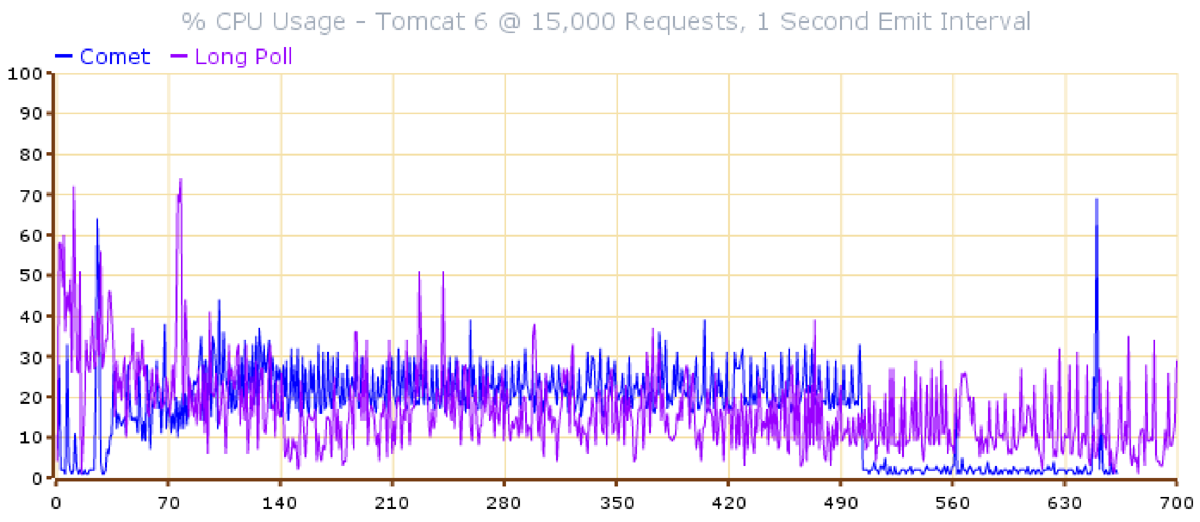


Figure 21: Tomcat 6 – Comet vs Long Poll – 1 Second Emit interval – CPU Usage.








% CPU Usage - Tomcat 6 @ 15,000 Requests, 1 Second Emit Interval
— Comet
— Long Poll
100
90
80
70
60
50
40
30
20
10
0
0
70
140
210
280
350
420
490
560
630
700

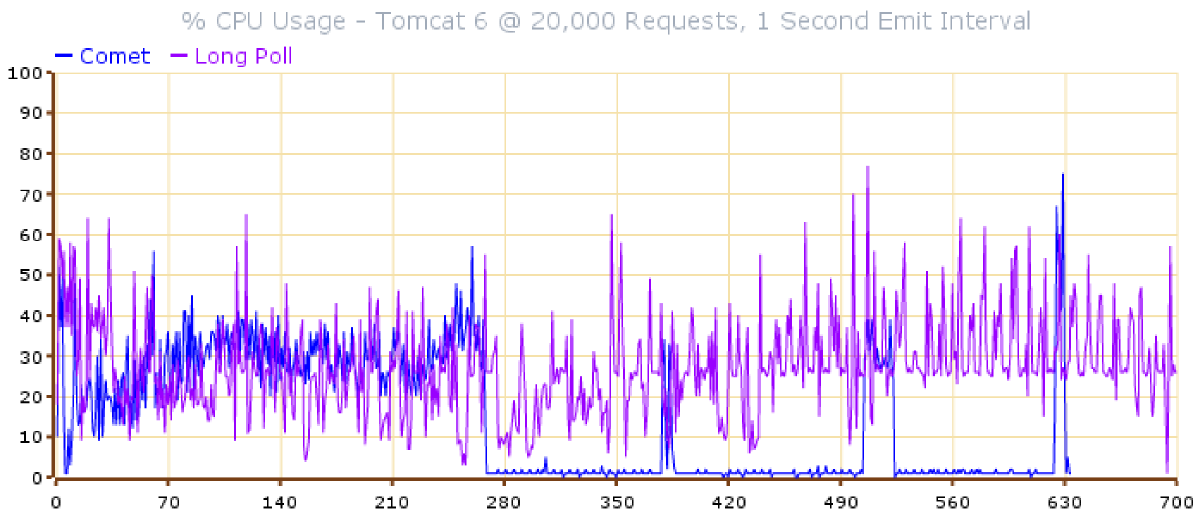
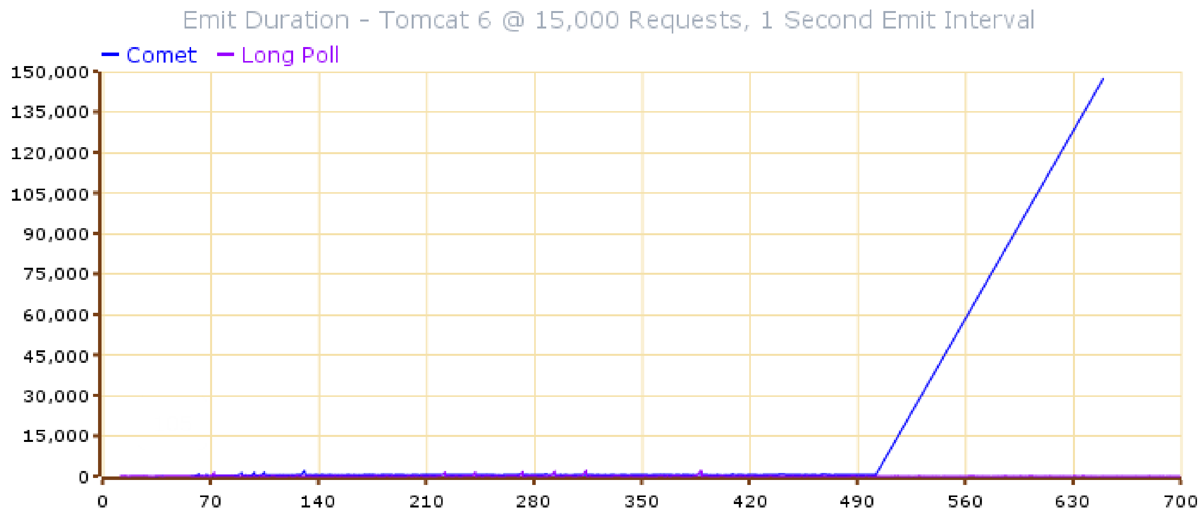

% CPU Usage - Tomcat 6 @ 20,000 Requests, 1 Second Emit Interval
— Comet
— Long Poll
100
90
80
70
60
50
40
30
20
10
0
0
70
140
210
280
350
420
490
560
630
700

Figure 22: Tomcat 6 – Comet vs Long Poll – 1 Second Emit interval – Emit Duration.



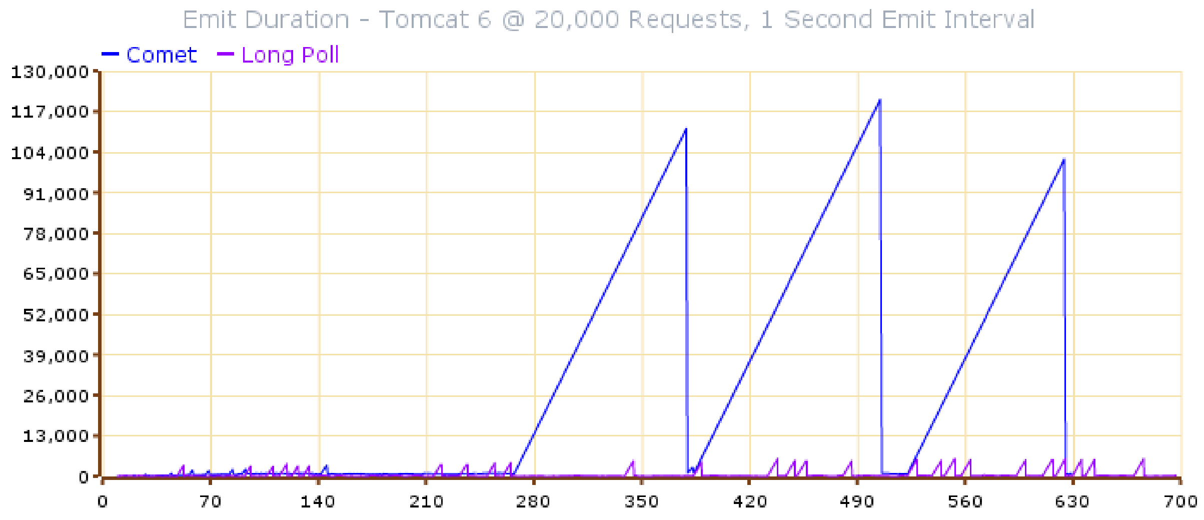
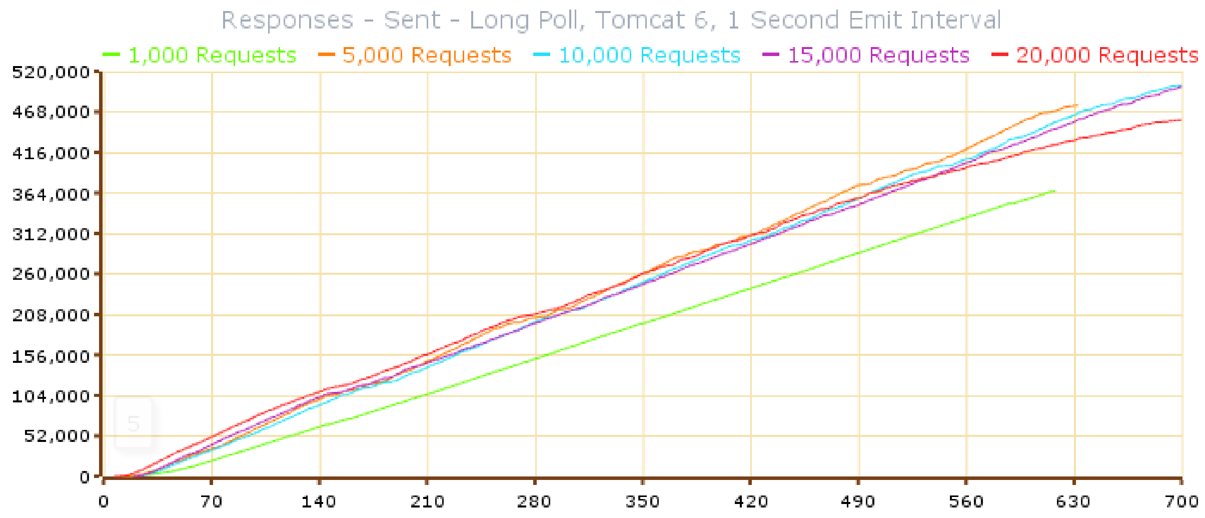
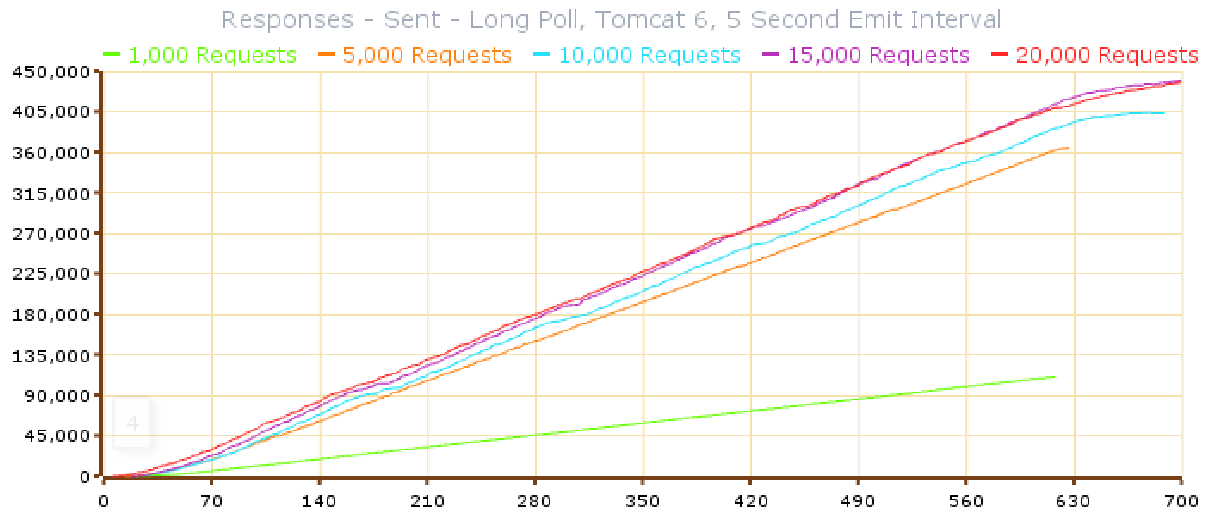
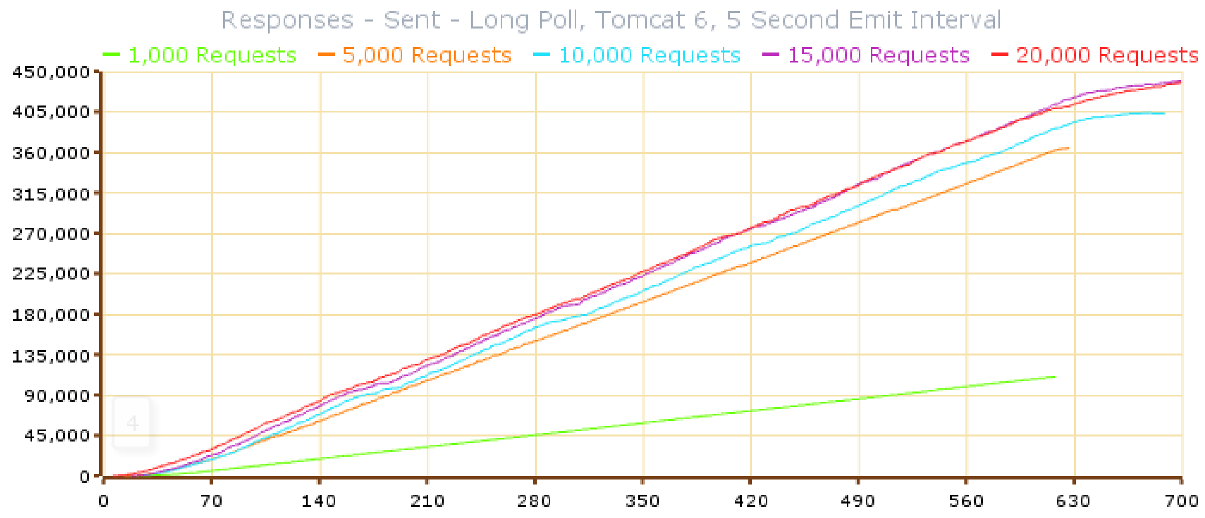


Figure 23: Tomcat 6 – Responses Sent – 1 to 30 Second Emit interval.







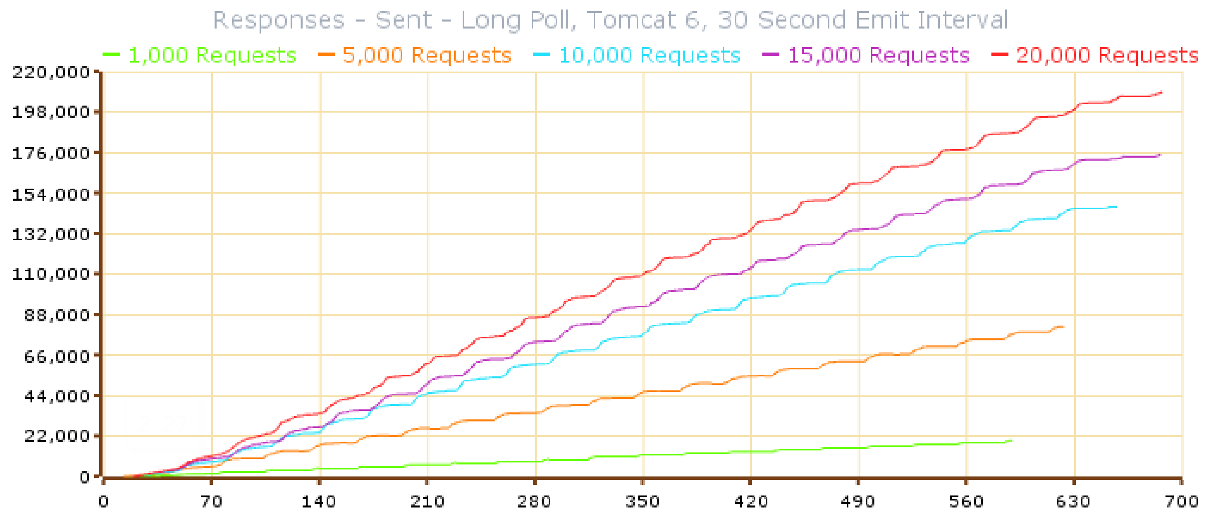
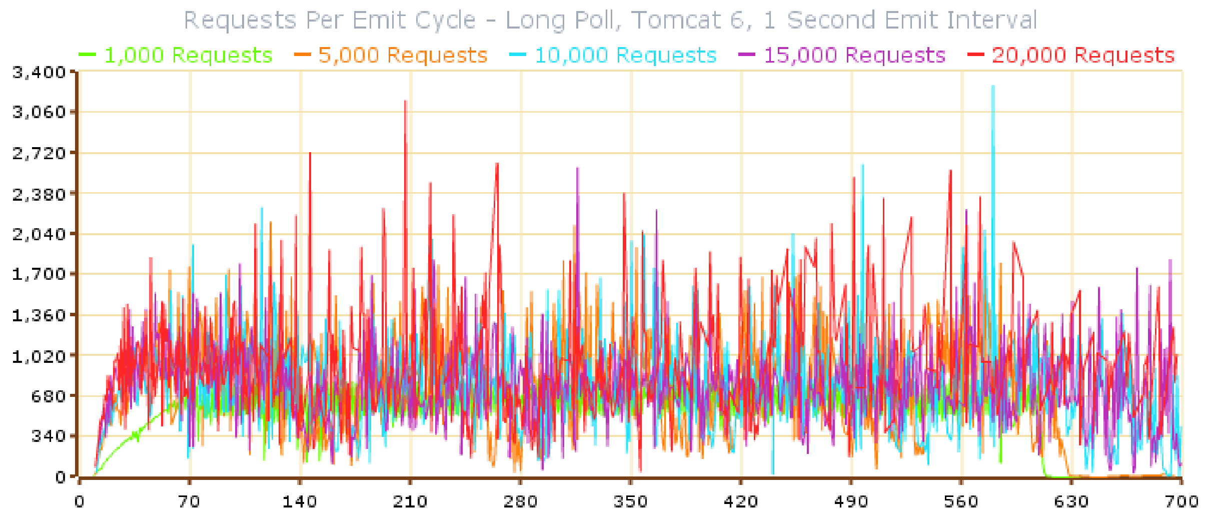

Responses - Sent - Long Poll, Tomcat 6, 30 Second Emit Interval
1,000 Requests
5,000 Requests
10,000 Requests
15,000 Requests
20,000 Requests
220,000
198,000
176,000
154,000
132,000
110,000
88,000
66,000
44,000
22,000
0
0
70
140
210
280
350
420
490
560
630
700

Figure 24: Tomcat 6 – Requests per Emit Cycle – 1 second and 30 second Emit Intervals.



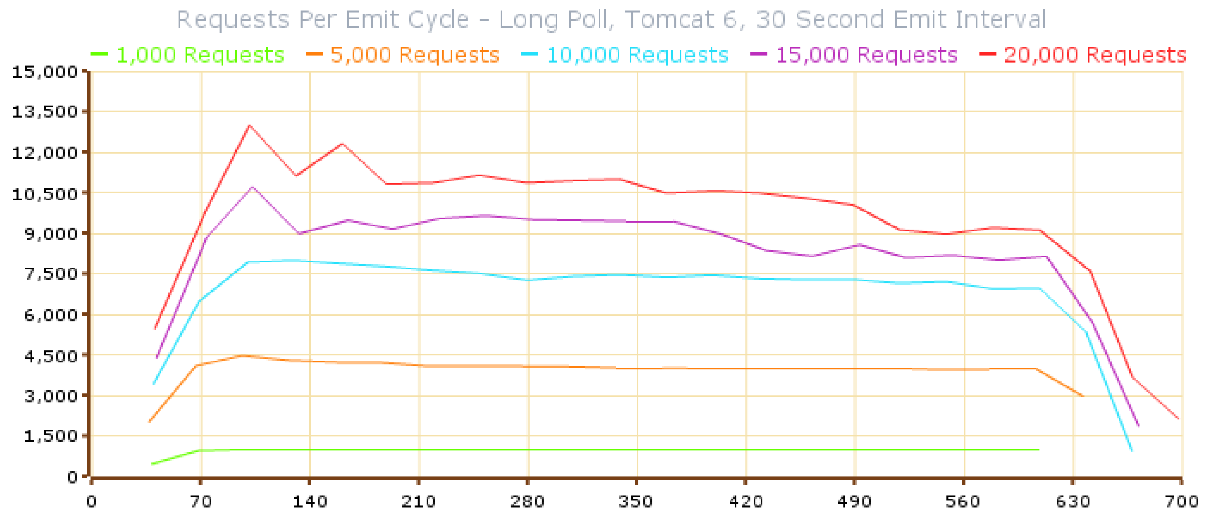
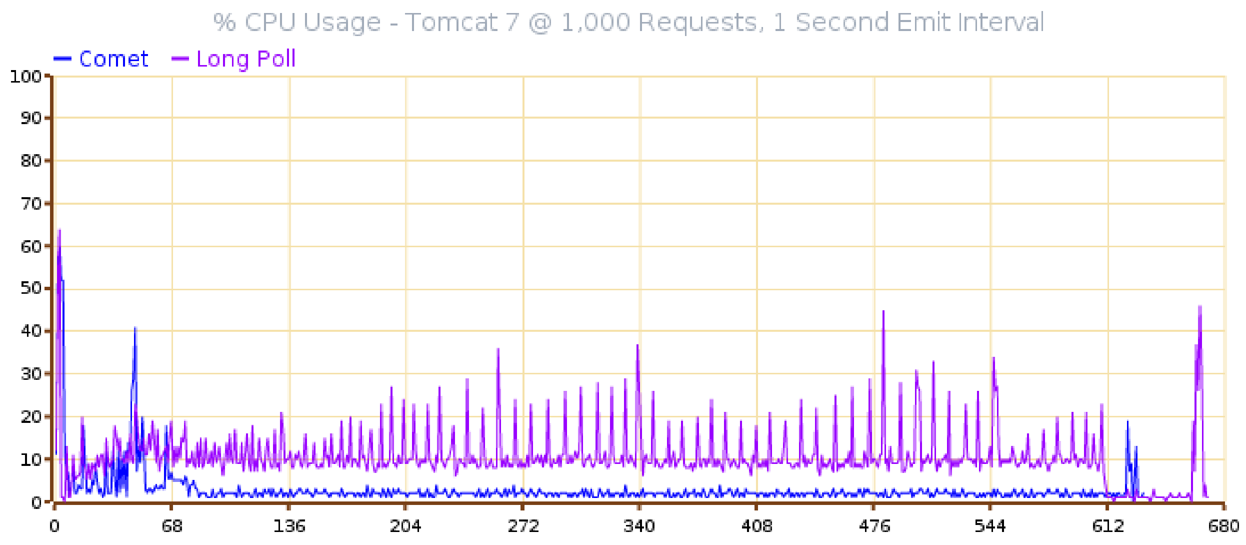
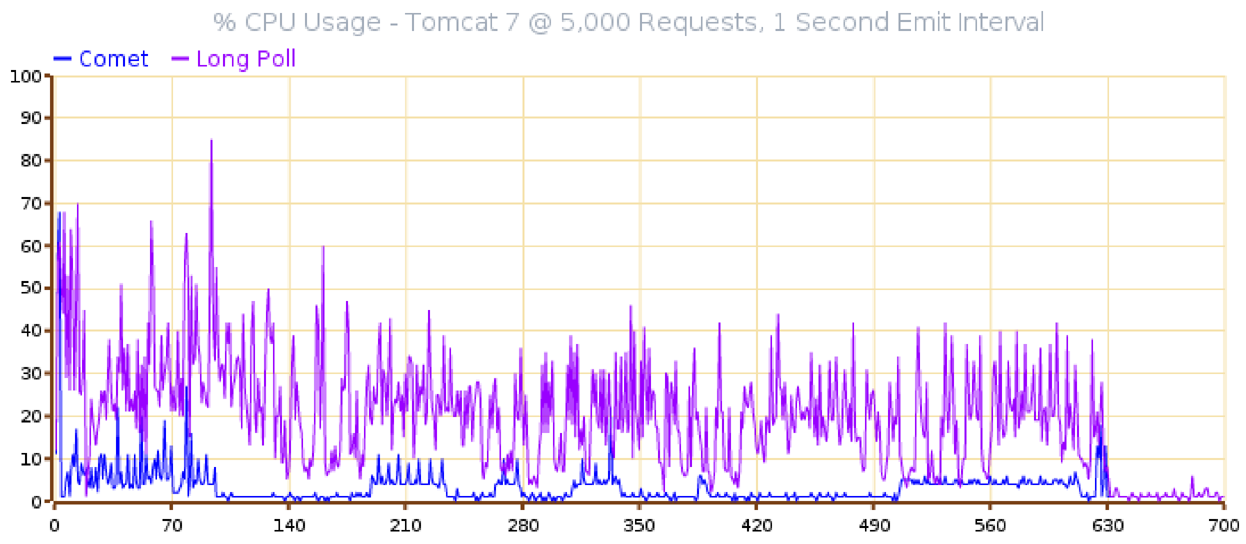
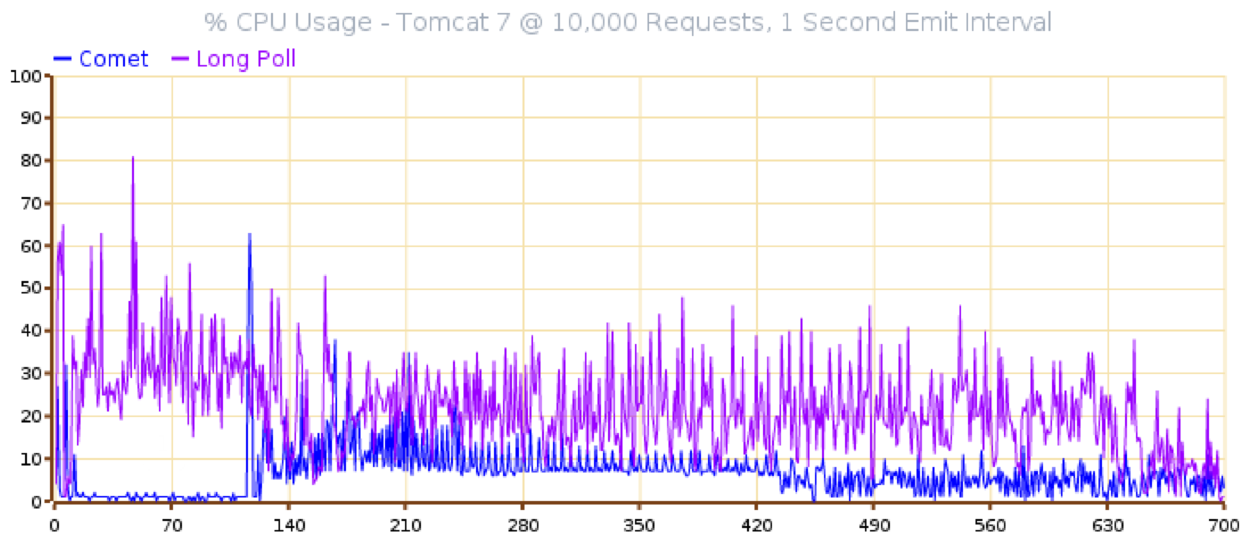
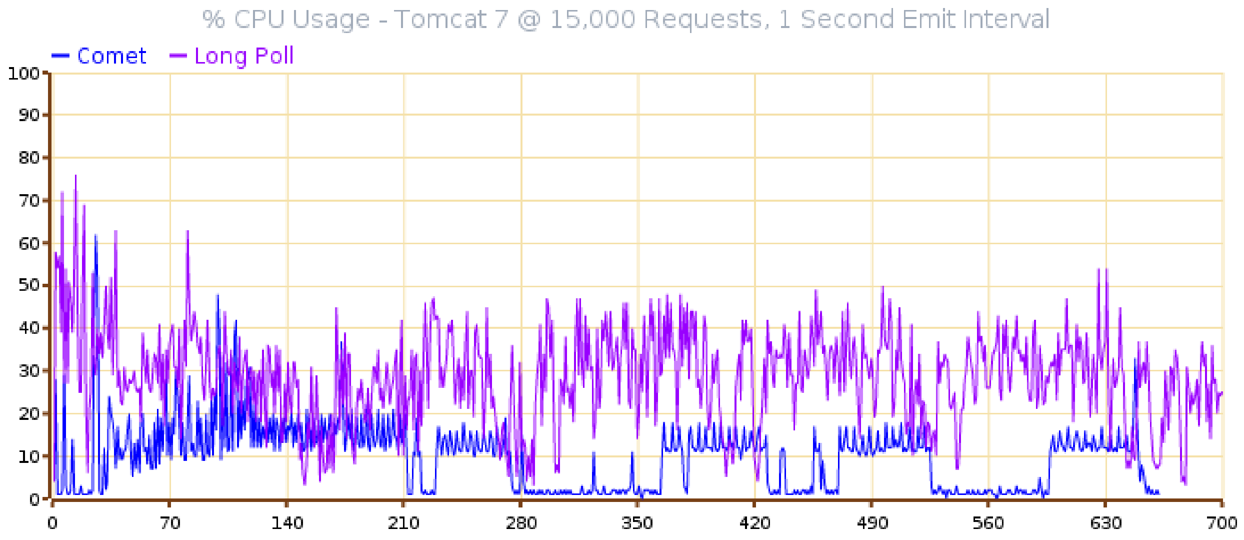


Figure 25: Tomcat 7 – Comet vs Long Poll – 1 Second Emit interval – CPU Usage.








% CPU Usage - Tomcat 7 @ 15,000 Requests, 1 Second Emit Interval
Comet
Long Poll
100
90
80
70
60
50
40
30
20
10
0
0
70
140
210
280
350
420
490
560
630
700

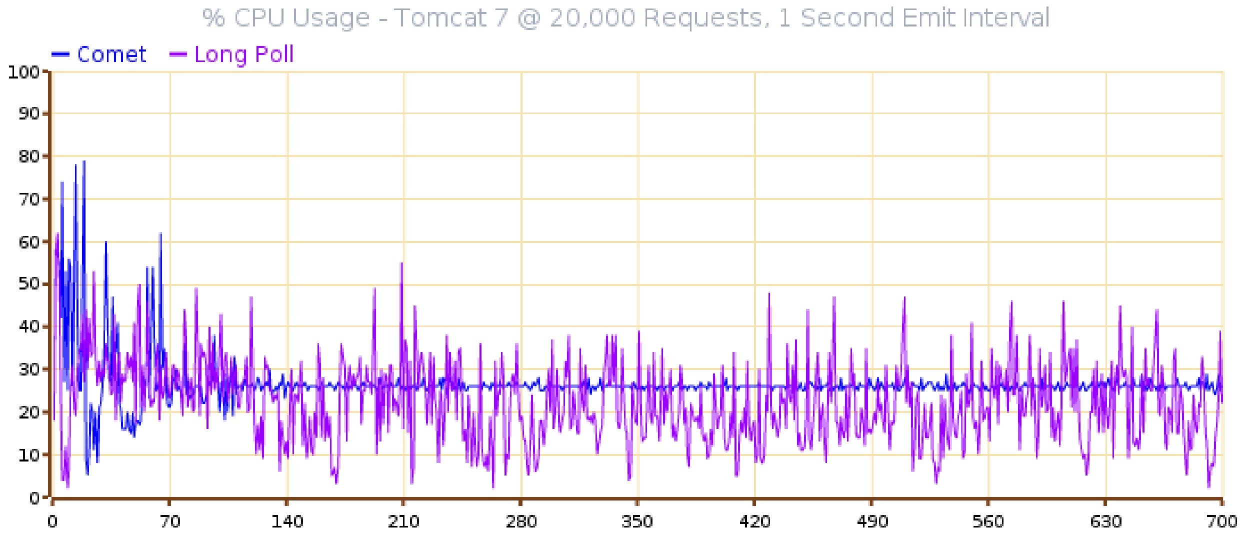
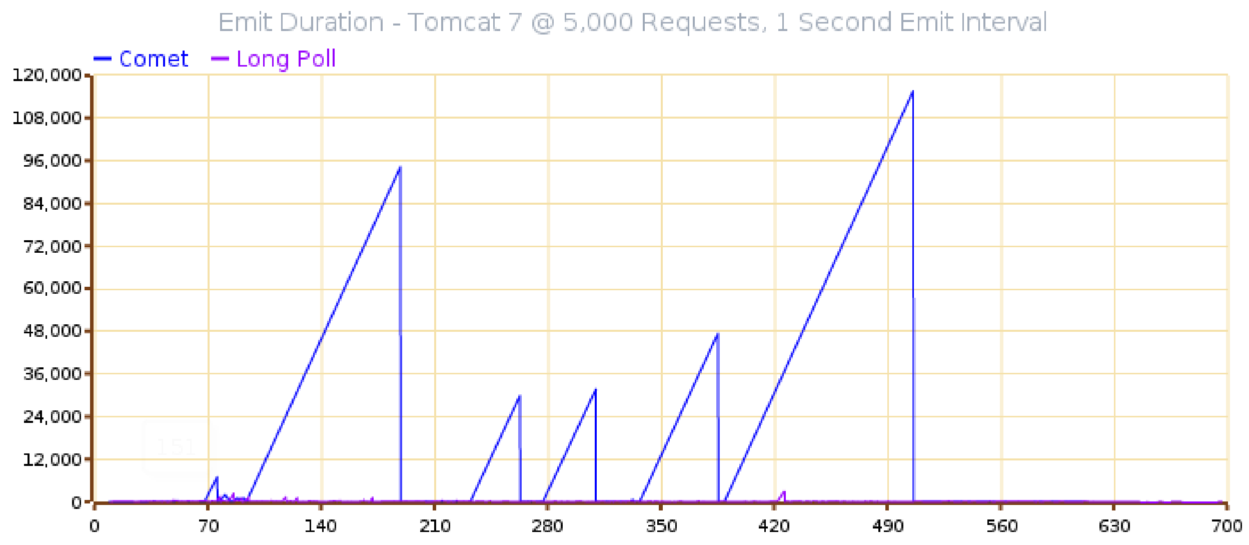

% CPU Usage - Tomcat 7 @ 20,000 Requests, 1 Second Emit Interval
Comet
Long Poll
100
90
80
70
60
50
40
30
20
10
0
0
70
140
210
280
350
420
490
560
630
700

Figure 26: Tomcat 7 – Comet vs Long Poll – 1 Second Emit interval – Emit Duration.



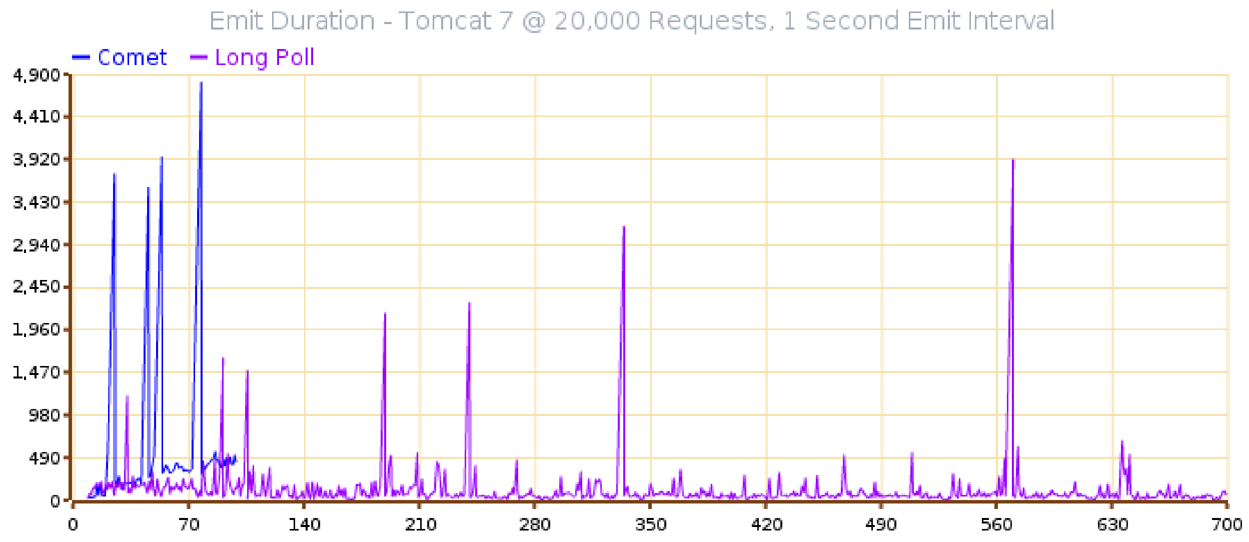
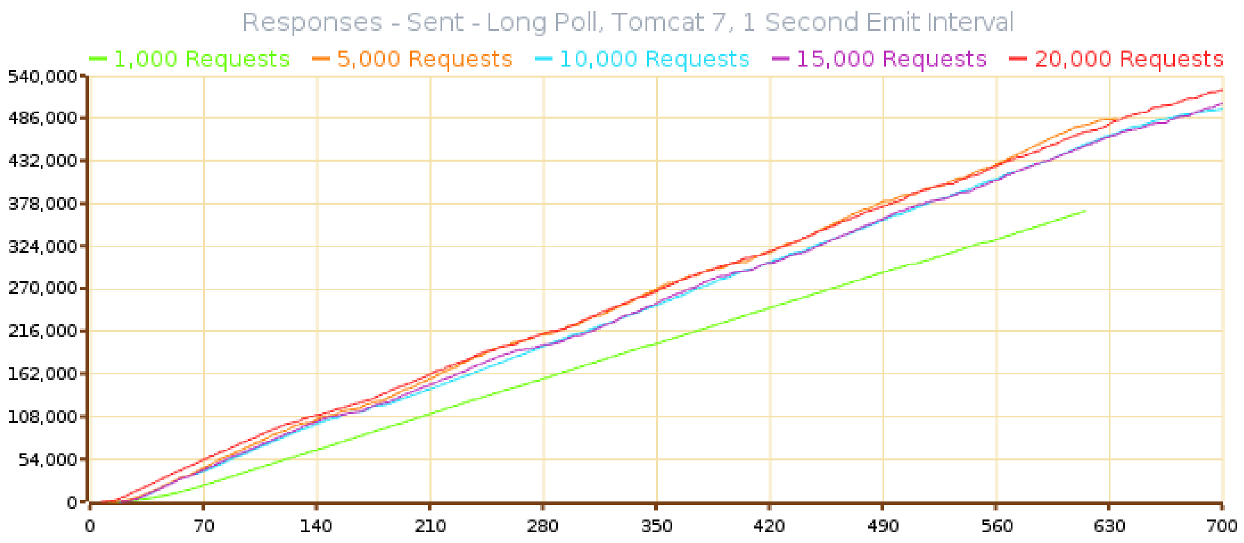
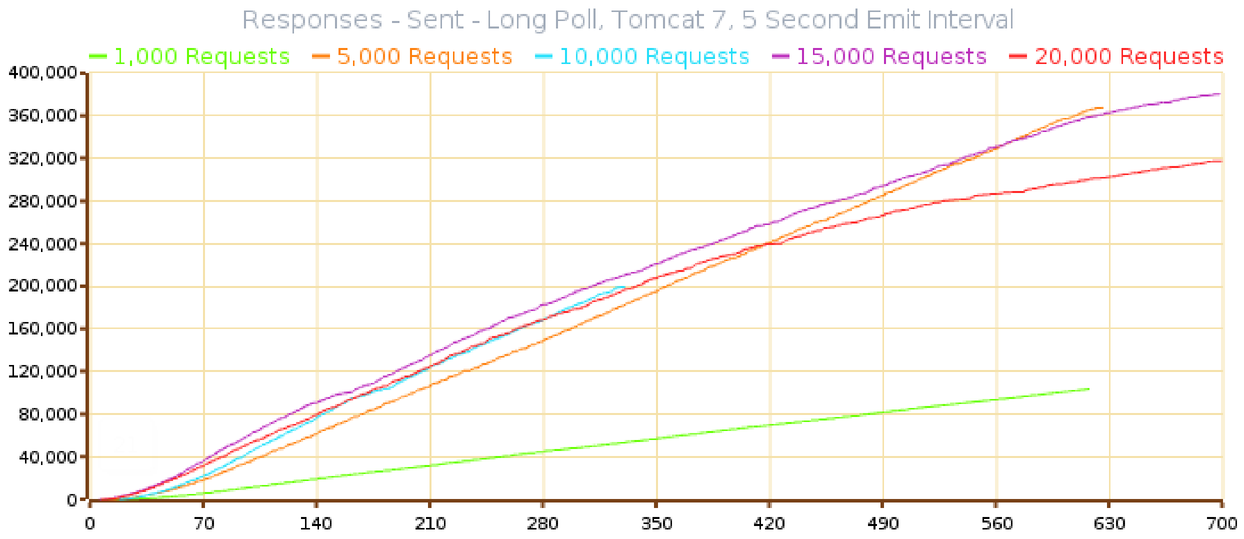
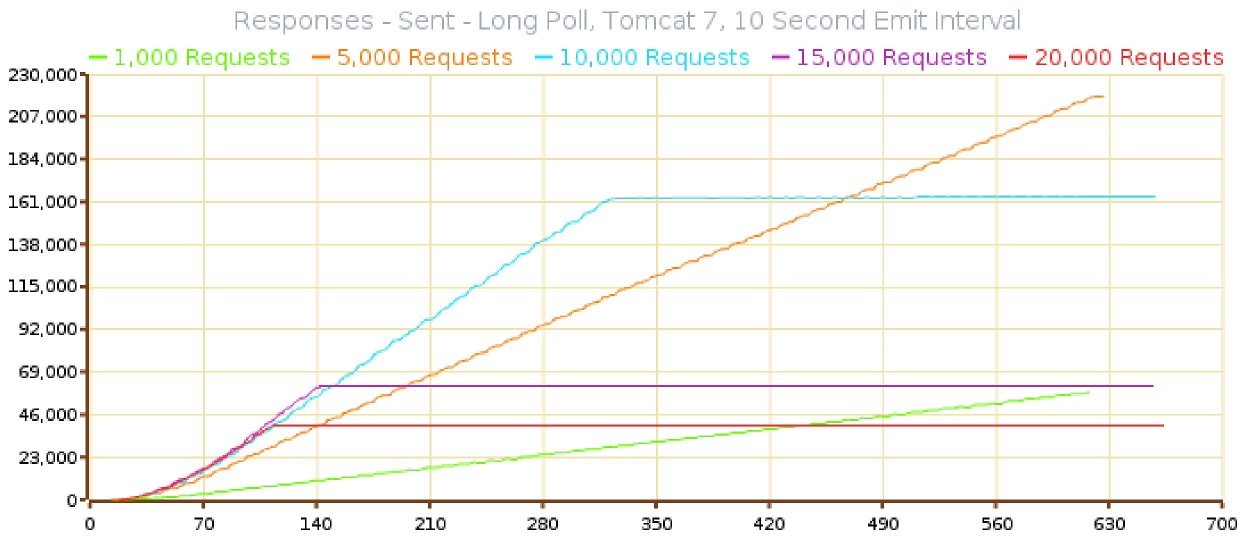


Figure 27: Tomcat 7 – Responses Sent – 1 to 30 Second Emit interval.







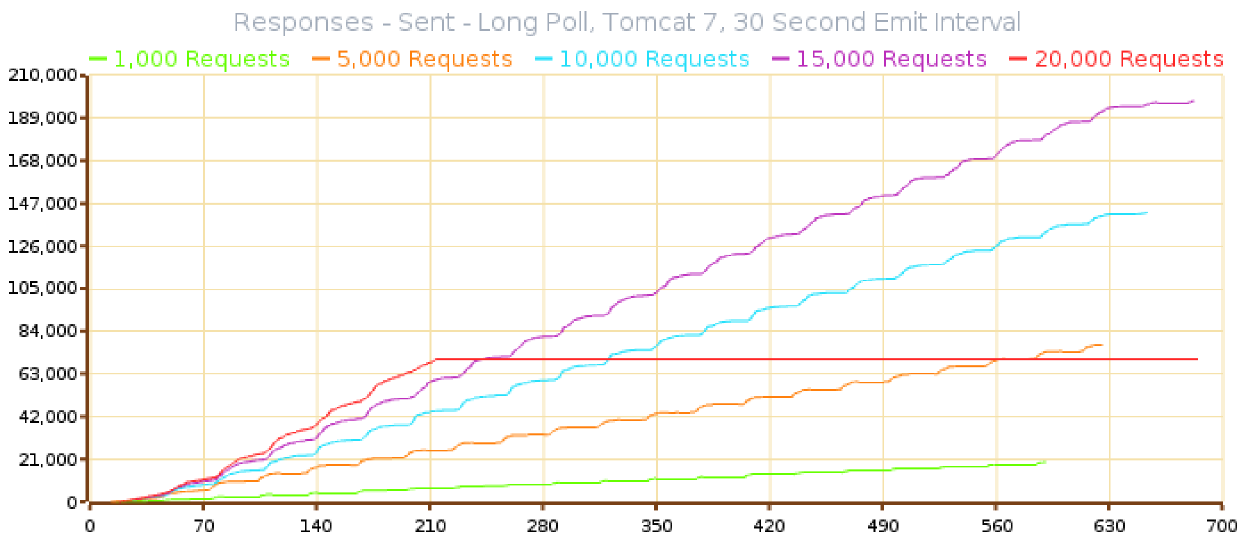
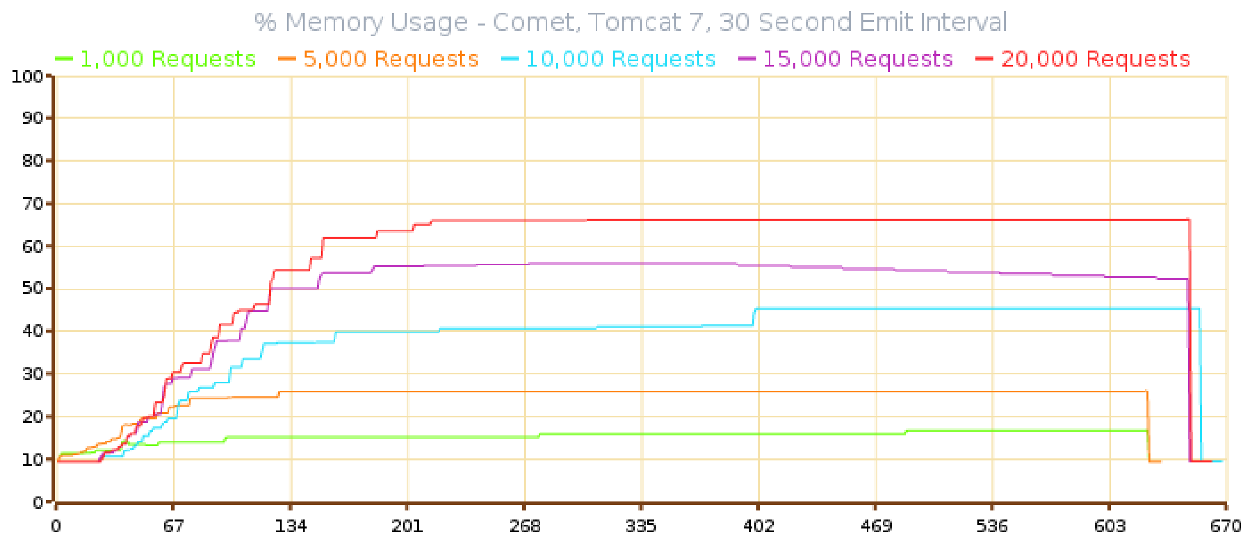
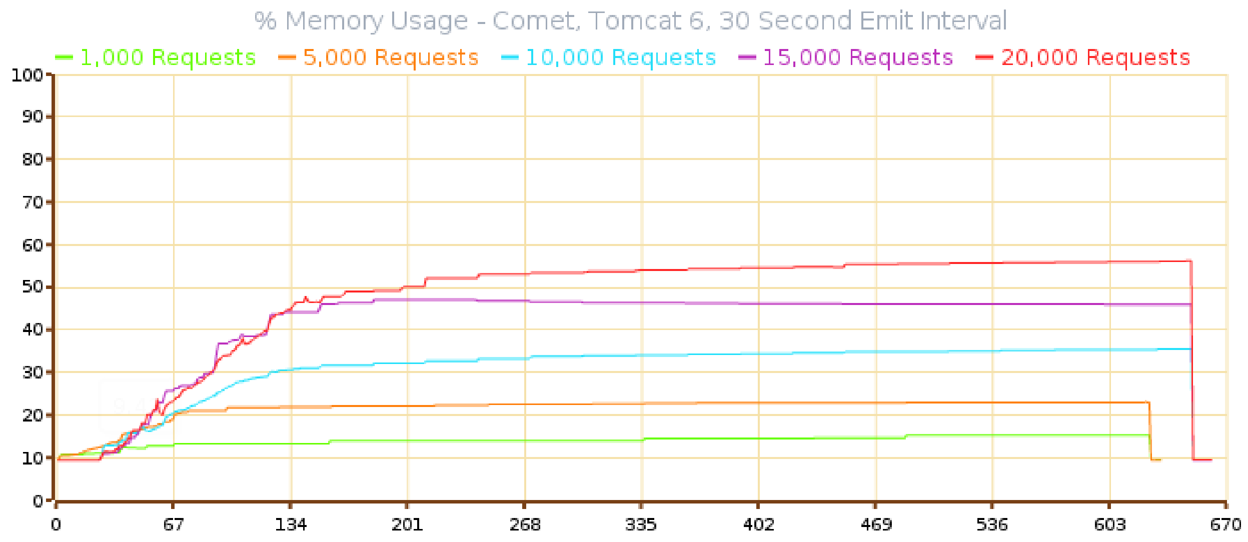
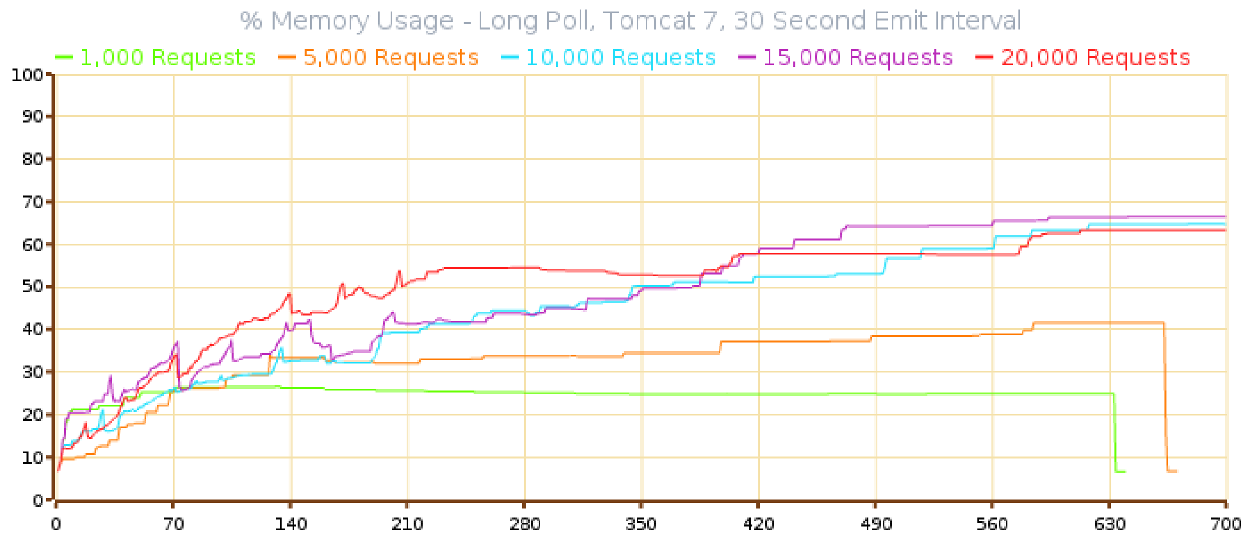
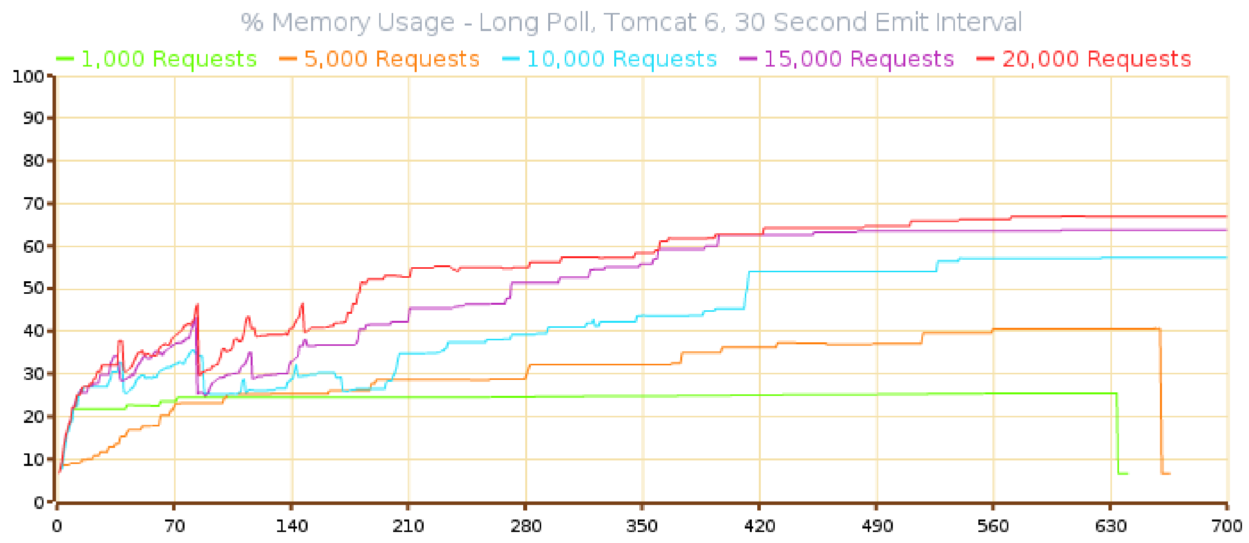

Responses - Sent - Long Poll, Tomcat 7, 30 Second Emit Interval
1,000 Requests
5,000 Requests
10,000 Requests
15,000 Requests
20,000 Requests
210,000
189,000
168,000
147,000
126,000
105,000
84,000
63,000
42,000
21,000
0
0
70
140
210
280
350
420
490
560
630
700

Figure 28: Tomcat 6, Tomcat 7 – 30 Second Emit Interval – Comet, Long Poll – Memory Usage.








% Memory Usage - Long Poll, Tomcat 6, 30 Second Emit Interval
1,000 Requests
5,000 Requests
10,000 Requests
15,000 Requests
20,000 Requests
0
10
20
30
40
50
60
70
80
90
100
0
70
140
210
280
350
420
490
560
630
700