

# Stress Testing Cometd

## Stress Testing Cometd (Jetty implementation)

These instructions show you how to stress test cometd from jetty 6.1.9 running on unix. The same basic steps apply to running on windows or mac and I'd be happy to add details instructions if somebody wants to contribute them.

The basic steps are:

1. configure/tune operating system of test client and server machines
2. install, configure and run jetty server
3. run jetty bayeux test client
4. interpret the results.

### Configure/Tune operating system.

The main change needed to the operating system is that it needs to be able to support the number of connections (== file descriptors) for the test on both the server machine and the test client machines needed.

For a linux system, the file descriptor limit is change in the `/etc/security/limit.conf` file.

Add the following two lines (or change any existing `nofile` lines):

```
* hard nofile 40000
* hard nofile 40000
```

There are many other things that can be tuned in the server stack, and the [zeus ZXTM](#) documentation gives a good overview.

### Install, configure and run Jetty

Jetty installation is trivial. See [Downloading Jetty](#), [Installing Jetty-6.1.x](#) and [Running Jetty-6.1.x](#).

For the purposes of cometd testing, the standard configuration of jetty (`etc/jetty.xml`) needs to be edited to change the connector configuration for:

- increase the max idle time
- increase the low resources connections.

The relevant updated section is:

```
<Call name="addConnector">
  <Arg>
    <New
class="org.mortbay.jetty.nio.SelectChannelCo
nnector">
      <Set name="host"><SystemProperty
name="jetty.host" /></Set>
      <Set name="port"><SystemProperty
name="jetty.port" default="8080"/></Set>
      <Set
name="maxIdleTime">300000</Set>
      <Set name="Acceptors">2</Set>
      <Set name="statsOn">>false</Set>
      <Set
name="confidentialPort">8443</Set>
      <Set
name="lowResourcesConnections">25000</Set>
      <Set
name="lowResourcesMaxIdleTime">5000</Set>
    </New>
  </Arg>
</Call>
```

Jetty comes with cometd installed in `$JETTY_HOME/webapps/cometd.war`.  
To run the server with additional memory needed for the test, use:

```
java -Xmx2048m -jar start.jar etc/jetty.xml
```

You should now be able to point a browser at the server at either:

- <http://localhost:8080/>

- `http://yourServerIpAddress:8080/`

Specifically try out the cometd chat room with your browser to confirm that it is working

## Run jetty bayeux test client

The jetty cometd bayeux test client generates load simulating users in a chat room. To run the client:

```
cd $JETTY_HOME/contrib/cometd/client
bin/run.sh
```

 depending on the version you might need to create a lib/cometd directory and put the cometd-api, cometd-java-server and cometd-java-client in it

- lib/cometd/
- lib/cometd/cometd-api-1.0.beta9.jar
- lib/cometd/cometd-client-6.1.19.jar
- lib/cometd/cometd-server-6.1.19.jar

The client has a basic text UI that operates in two phases: 1) global configuration 2) test runs. An example global configuration phase looks like:

```
# bin/run.sh
2008-04-06 13:43:57.545::INFO: Logging to
STDERR via org.mortbay.log.StdErrLog
server[localhost]: 192.126.8.11
port[8080]:
context[/cometd]:
base[/chat/demo]:
rooms [100]: 10
rooms per client [1]:
max Latency [5000]:
```

The Enter key can be used to accept the default value, or a new value typed and then press Enter. The parameters are their meaning are:

- **server** - The host name or IP address of the server running Jetty with cometd
- **8080** - The port (8080 unless you have changed it in jetty.xml)
- **context** - The context of the web application running cometd (cometd in the test server).
- **base** - The base bayeux channel name used for chat room. Normally you would not change this.

- **rooms** - The number of chat rooms to create. This will combine with the number of users to determine the users per room. If you have 100 rooms and 1000 users, then you will have 10 users per room and every message sent will be delivered 10 times. For runs with >10k users, 1000 rooms is a reasonable value.
- **rooms per client** This allows a simulated user to subscribe to multiple rooms. However, as these are randomly selected, values greater than 1 will mean that the client will not be able to accurately predict the number of messages that will be delivered. Leave this at 1 unless you are testing something specific.
- **max Latency** If the latency for delivering a message is greater than this value (in ms), abort the test.

After the global configuration, the test client loops through individual tests cycles. Again Enter may be used to accept the default value. Two iterations of the test cycle are below:

```
clients [100]: 100
clients = 0010
clients = 0020
clients = 0030
clients = 0040
clients = 0050
clients = 0060
clients = 0070
clients = 0080
clients = 0090
clients = 0100
Clients: 100 subscribed:100
publish [1000]:
publish size [50]:
pause [100]:
batch [10]:
0011111111221111111111111111111110000000000000000
0000000000000000000000000000000000000000000000000
00000000000000

Got:10000 of 10000
Got 10000 at 901/s, latency min/ave/max
=2/41/922ms
--
```



```
Got 10000 at 972/s, latency min/ave/max
=3/26/172ms
--
```

The parameters that may be set are:

- **clients** - The number of clients to simulate. The clients are kept from one test iteration to the next, so if the number of clients changes on an incremental number of new clients are created or destroyed. (NB. currently reducing clients produces a noisy exception as the connection is retried. This can be ignored).
- **publish** - The number of chat messages to be published in this test. The number of messages received will be this number multiplied by the users per chat room (which is the number of clients divided by the global number of rooms).
- **publish size** - The size in bytes of the chat message to publish.
- **pause** - A period in ms to pause between batches of published messages.
- **batch** - The size of the batch of publish messages to send in a burst.

While the test is executing, a series of digits is output to show progress. The digits represent the current average latency in units of 100ms. So a 0 represent <100ms latency from the time the message was publish by the client to when it has been received on the client. 1 represents a latency >=100ms and <200ms etc.

At the end of the test cycle the summary is printed showing the total messages received, the message rate and the min/ave/max latency.

## Interpreting the results.

Before producing numbers for interpretation, it is important to run a number of trials and to allow the system to "warm up". During the initial runs, the java JIT compiler will optimize the code and object pools will be populated with reusable objects. Thus the first runs at a give number of clients is often slower, and this can be seen in the test cycle shown above where the average latency initially blew out to over 200ms before it was reduced back to <100ms. The average and max latency for the second run were far superior to the first run.

It is also important to use long runs for producing results, so that:

- Any statistical effect of the ramp-up and ramp-down periods in each test are reduced.
- So that any resources (queues, memory, file descriptors, etc) that are being used in a non-sustainable way will have a chance to max out and cause errors, garbage collections or other adverse affects.
- Any occasional system hiccups caused by other system events are included in the results

Typically it is best to start with short low volume test cycles and to gradually reduce the pause or increase the batch to determine approximate maximum message rates. Then the test duration can be extended by increasing the number of messages published or the number of clients (which also increases the message rate as there will be more users per room).

A normal run should report no exceptions or timeouts. For a single server and single test client with 1 room per simulated client, then the expected number of messages should always be received. If the server is running clustered, then as this demo has no cluster support, the messages will be reduced by the a factor equal to the number of servers. Similarly if multiple clients are used, each test client will see messages published from the other test client, so the number of messages received will be in excess.

## Load balancers

If you are testing a load balancer, then it is very important that there is affinity, as the bayeux client ID must be known on the worker node used and both connections from the same simulated node must arrive at the same worker node. However, the test does not use HTTP sessions, so any cookies used for affinity will need to be set by the balancer (the test client will handle set cookies).

**i Reality check**

In reality, IP source hash would be a sufficient affinity for bayeux, but in this test, all clients come from the same IP. Also note that the real dojo cometd client has good support for migrating to new nodes if affinity fails or the cluster changes. Also a real chat room server implementation would probably be backed by JMS so that multiple nodes would still represent a single chat space.

If you are testing a load balancer, then you should start with a cluster of 1, so that you can verify that no messages are being lost. Then increase the cluster size and be content that you will not have exact message counts and must adjust by the number of nodes.