



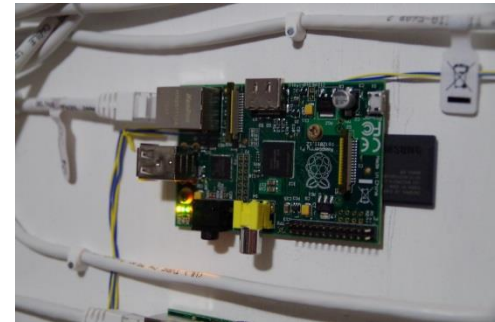
Performance Testing with a Raspberry Pi Wall running Java

Erik Wramner
CodeMint AB
<http://codemint.com>



This session describes how we built a Raspberry Pi wall and used it for affordable and realistic load tests

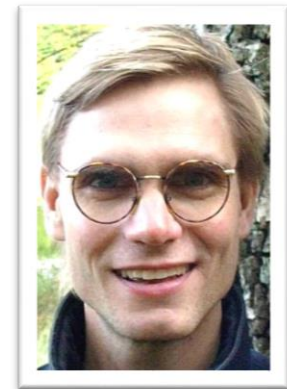
- Background
- Alternatives
- Implementation
- Load tests
- Live demo
- Results
- Questions



Background

Erik Wramner has worked with Java since 1997 and has developed and tuned network applications for a long time

- Erik has been an active Java developer since 1997 and wrote distributed network applications even earlier in C/C++
- Performance has often been a key requirement
 - Cross-border stock trading gateway
 - Credit card payment backend
 - Telecom SS7 recorder
 - Delivery chain replenishment & forecasting
 - ...
- Erik has worked as a performance expert internationally, helping failing projects to get back on track



CodeMint is a Swedish consulting company with an edge in Java, Oracle's technology stack and performance tuning

- Java is our platform of choice – we have built many mission-critical applications in Java, particularly for the finance industry
- We have worked extensively with Oracle's technology stack



- Performance tuning is often included when we build applications – in addition we perform health checks and help customers test and tune existing systems

In one assignment we helped develop a business critical network server with high uptime and performance targets

- The application had to be available 24x7x365, no downtime
- The expected load was fairly high with perhaps 50.000 concurrent long-lived connections and up to 100 business transactions per second
- The server used a custom binary protocol and SOAP
- Security requirements were strict

How could we ensure that it worked as expected?

Alternatives

We decided to run prolonged load tests and considered several alternatives before settling on a “Raspberry Wall”

- Single server
- Developer workstations
- The cloud
- ...and finally the Raspberry Wall

The most common approach is probably to use a single server¹ for load tests, but that is far from ideal

- A *dedicated* server is expensive and hard to obtain in many organizations, in particular when time is short
- A *shared* server may not have the resources necessary for proper long-running and intensive load tests
- A single server does not generate a realistic work load
 - ✓ If the tests run on the application's own server they consume resources (skewing the results) and use the loopback² interface
 - ✓ If the tests run on another server they are still likely to use a single network interface

Note 1: Usually the build server or the application's own server or a single developer workstation.

Note 2: A simple Grinder test on Windows 7 runs more than ten times faster when the network cable is removed! This is dodgy.

Another alternative is to use several developer workstations for distributed load tests, but again there are problems

- In some organizations the developers are not allowed to install software on their own machines
- In many organizations the developer workstations are rebooted regularly for automated upgrades, that would break the tests
- Developer productivity and morale will suffer if their (perhaps already slow) machines get overloaded by tests
- Developer machines are typically heavily utilized every now and then, it can be tricky to generate a consistent load profile over time

Cloud-based load tests can be a big win, but in our case they were simply not an option

- Cloud-based load tests have many advantages in general
 - ✓ They are quick to setup for simple applications
 - ✓ They can generate realistic load from many remote sites
 - ✓ They come loaded with canned reports and analysis tools
 - ✓ They can be very cost-effective for short tests
- Unfortunately they were not an option in this case
 - ✓ They could not be allowed through the firewall
 - ✓ They are obviously expensive¹ for long-running tests with many concurrent users
 - ✓ They can sometimes support custom protocols, but “up and running in five minutes” assume simple HTTP tests

Note 1: We needed 40-50.000 concurrent users and wanted to run the tests repeatedly for months or at least weeks. There are many variables. With the free load tester from [webperformance.com](#) and Amazon EC2 load generators at \$1 per hour and generator and the recommended 25 generators for 50.000 users it would cost \$16.800 per month. Most vendors publish no price at this level (call for options). For example [blazemeter.com](#) refers to the Enterprise option (no published price), but they charge \$1499 per month for 30 test hours with 40.000 users. You get the idea.

Finally, we got the idea to build a wall with cheap Raspberry Pi units in order to run realistic load tests inside the firewall

- The wall can run inside the firewall and can be isolated from all systems except for the test server
- The wall can be built cheaply with the right design
- The Raspberry Pi runs Java, so it is easy to write or reuse existing tests that support custom protocols
- The tests can be realistic with many physical network cards, many comparatively slow clients and long test durations

Last but not least, it seemed fun!

For reference the Raspberry Pi is about as fast as a 300MHz Pentium 2, but with much better graphics

- ARM1176JZFS 700MHz
- 512M RAM
- 10/100M Ethernet
- SD card for storage
- No real time clock



Source: <http://www.raspberrypi.org/faqs>, 2013-09-04.

Implementation

We came up with a cost-effective design and built a small prototype – then we built and configured a large wall

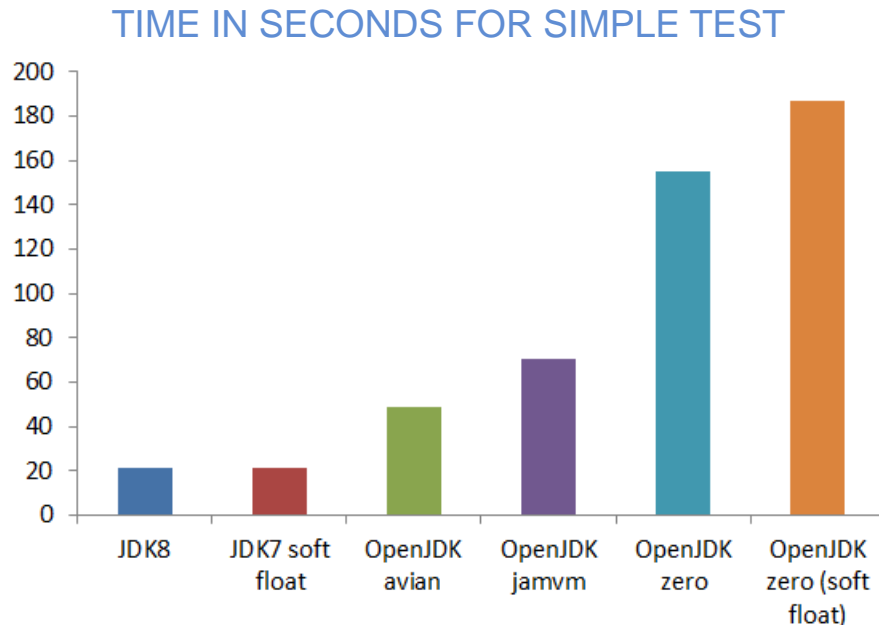
- Design
- Construction
- Installation and configuration
- Framework support
- Cloning and individualization

The Raspberry Pi itself is cheap, peripherals cost – the key is to avoid them and to find a good cost effective network switch

- A Raspberry Pi with 512M RAM costs \$25-\$40, but it is easy to double that with memory, network cables, USB cables, power and so on
- We soldered the units directly to a power supply – for small to medium walls with less than 35 units an old PC power supply will do
- We bought memory cards for about \$10 per unit
- We bought network cables – that added about \$10 per unit, but it saved us enough time to make it well worth it
- The switch is the most expensive part – it must have at least one gigabit port towards the LAN and one 100Mbit port per Raspberry Pi
 - ✓ Small consumer switches are cheap, but expensive per unit
 - ✓ Large switches tend to have Enterprise price tags
- We ended up at about \$3000 for 48 units

We had to consider what Java version to use – the choice was between free and fast, but Today that is a no-brainer

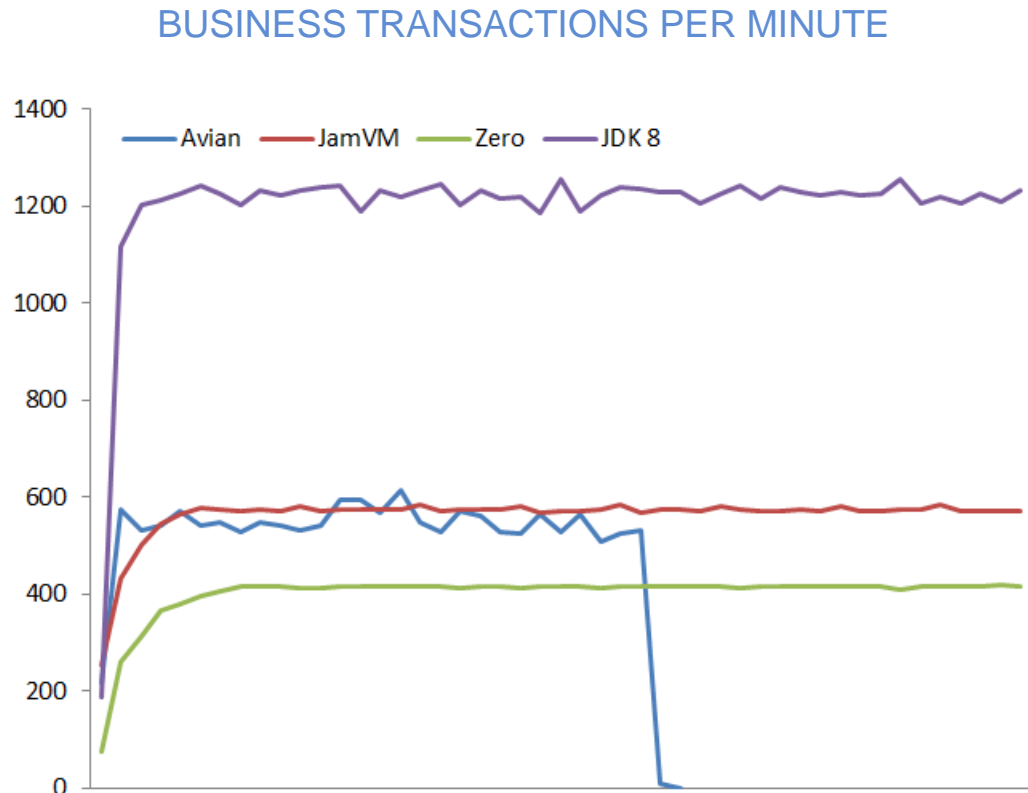
- Java SE 8 Embedded (early release) supported hard float and was the fastest out there, but it could not legally be used in production
- Java SE 7 Embedded was fast, but required soft float and had to be licensed for production



- OpenJDK was free and supported hard float, but it was much slower – in particular with the default Zero engine
- Starting from September 2013. Raspbian ships with a version of Oracle's Java SE 7 that supports hard float, so Today the recommendation is simple – go with Oracle!

Oracle's JVM still rules with more realistic tests, so it is really good news that it is included without any fuss

- The Oracle JVM was twice as fast as the closest contender – in addition the load was lower, indicating spare capacity!
- With Avian we got timeout errors and irregular throughput, probably due to garbage collection¹
- With Avian the JVM frequently died and had to be restarted after an hour or so
- JamVM and Zero performed well with stable throughput and without any errors



Note 1: Avian requires memory options to be expressed in bytes, so 256M is not understood. Even with correct options the throughput varies a lot, though, see chart.

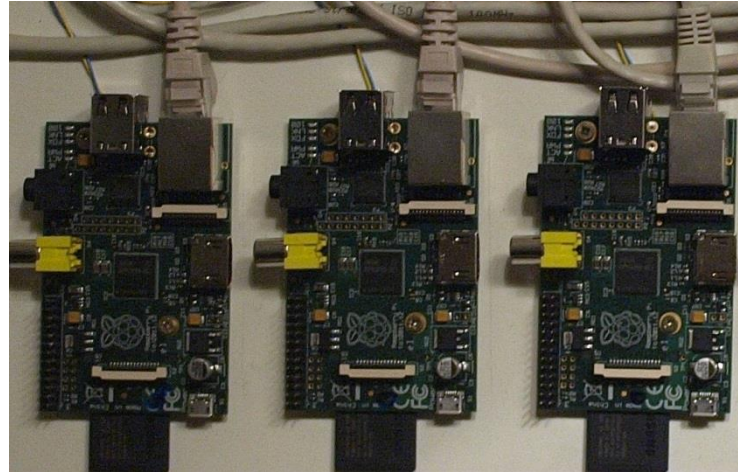
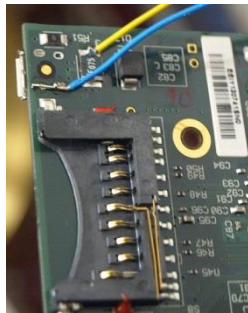
Note that recent Raspberry Pi units fail to boot with the soft-float image – fortunately there is a solution

- When we built the wall Java SE 7 Embedded was the only supported Java version from Oracle and it worked only with the old soft float image
- Recent Raspberry Pi units with the Hynix¹ memory chip cannot boot with the old soft-float image (we encountered this, some units worked and some seemed broken with the same memory card – nasty!)
- Luckily the old kernel works if it gets past the boot sequence – if the boot files are updated from a recent image the old image can be used
 - ✓ Write the soft float image to a memory card
 - ✓ Write the latest hard float image to another card or device and copy bootcode.bin, start.elf and fixup.dat from the boot partition to the boot partition for the soft-float image – copying only start.elf as some blog posts suggest is not enough
 - ✓ The updated soft float image should boot and everything should work as expected



Note 1: The black, square RAM chip in the middle of the board – if it says Hynix or Samsung the unit will probably not boot with soft float.

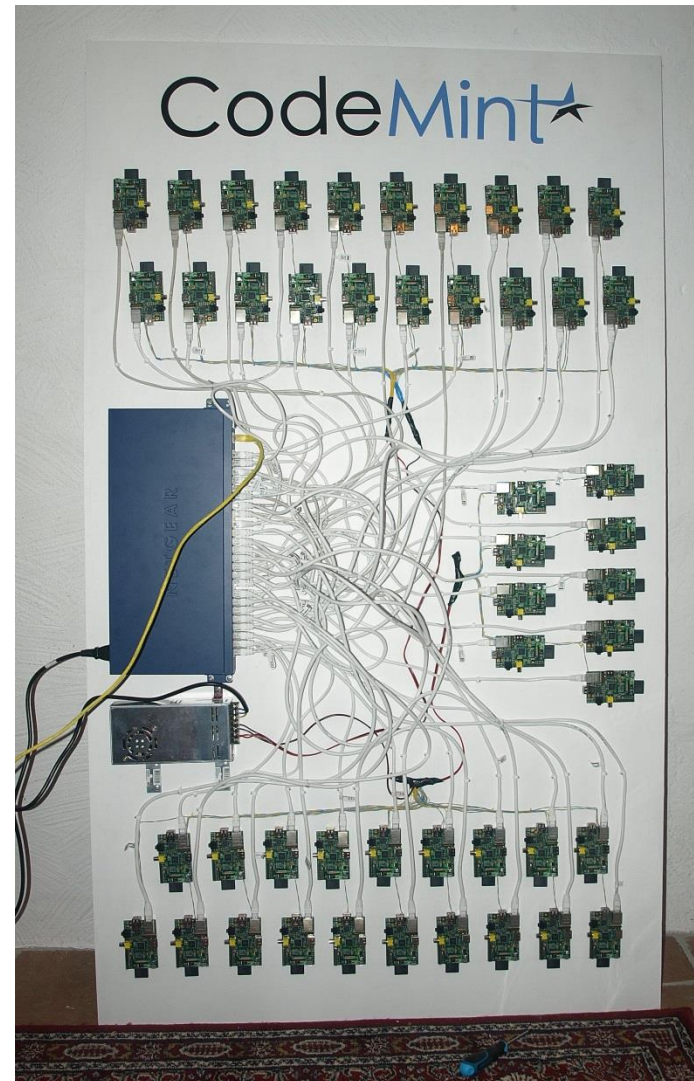
We built a prototype using spare parts in order to make sure that we were on the right track



It worked, but could not generate 100 business transactions per second with SSL, so we had to build a larger wall

We built the real wall with 48 Raspberry Pi units soldered to a 5V power supply using a network switch for small businesses

- The network switch has 48 100Mbit ports and two 1Gbit ports, perfect for our application
- The power supply has ample margins and can easily support 48 units
- The Raspberry Pi units are soldered to the power supply, no expensive USB cables are used



The installation is straightforward with the latest hard-float Raspbian image

- Download the latest hard-float “wheezy” image from <http://www.raspberrypi.org/downloads> and write it to a card

```
sudo dd if=2013-05-25-wheezy-raspbian.img of=/dev/sdb bs=1M
```

- Boot from the card and connect as pi/raspberry with SSH
- Perform initial configuration (`sudo raspi-config`)
 - ✓ Expand the file system (or intentionally keep it small)
 - ✓ Disable boot to desktop
 - ✓ Overclock (we went for “Medium”)
 - ✓ Advanced options, memory split: 16M for GPU
- Optionally register a custom NTP server in `/etc/ntp.conf`
- Reboot, update installed packages and install Java

```
sudo apt-get update; sudo apt-get upgrade  
sudo apt-get install oracle-java7-jdk
```

It is a good idea to use high-quality SD cards and to keep the number of writes down for the long haul

- We bought cheap (class 4) memory cards and in hindsight that was stupid – we have had 15 corrupted file systems and four broken SD cards in less than a year
- Disable swapping (just in case)

```
sudo dphys-swapfile swapoff  
sudo dphys-swapfile uninstall  
sudo update-rc.d dphys-swapfile remove
```

- Remove unwanted cron jobs

```
cd /etc; sudo rm cron.weekly/man-db cron.daily/bsdmainutils cron.daily/man-db  
cron.daily/aptitude cron.daily/apt
```

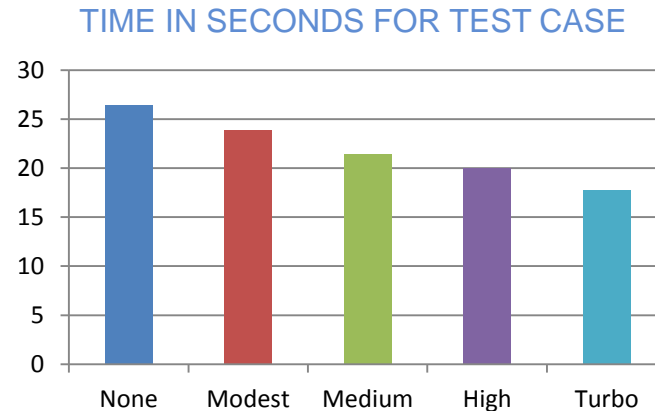
- Move the logs to a RAM disk and disable atime (/etc/fstab)

```
none /var/log tmpfs size=5M,noatime 0 0
```

- Limit logging (/etc/rsyslog.conf, don't fill up the RAM disk)

Overclocking improves performance, but beware as it may cause instability and corrupted SD-cards

- The raspi-config tool has five pre-defined levels for overclocking – it is also possible to fine-tune the settings manually
- Overclocking does make a difference, as the graph shows
- Overheating is not an issue – at least not in Sweden!
- There are many reports of corrupted memory cards with “Turbo”, even using class 10 cards – I would not go higher than the “High” level and in fact we picked “Medium”



The Raspberry Pi probably supports your load test framework of choice, but consider the CPU and memory limitations

- We had existing tests for The Grinder 3 (one of the leading Java load test frameworks where tests are written in Jython) and they worked out of the box – in general this is probably the way to go, especially for HTTP, as it makes the tests more portable
- The Raspberry is much slower than a PC and it has a fairly limited amount of memory – we wrote our own tailored load test framework in order to reduce unnecessary overhead
 - ✓ No Jython engine
 - ✓ Fewer classes, smaller footprint
 - ✓ Several simulated clients per thread (think times), making it easier to translate the results into business terms and reducing the number of context switches

Installation and configuration takes time – do it once, then clone the memory card and script individualization afterwards

- Install and test everything with a single Raspberry Pi
- If you are using soft float (no longer likely), make sure that all Raspberry Pi units support it or update the boot files as described earlier in this presentation
- Create an image from the SD card
- Copy the image to all the other cards
- Start the wall and optionally apply individual settings such as unique host names using scripted SSH

The image also serves as an excellent backup, just in case!

Load tests

A load test should determine if the application meets its performance requirements – they should be measurable and measured

- Determine targets (requirements)
- Generate load (Raspberry Wall)
- Measure
 - ✓ Client-side statistics
 - ✓ Server-side statistics
 - ✓ Performance monitors (jconsole, perfmon, top)
 - ✓ Profiling (visualvm)
- Analyze results

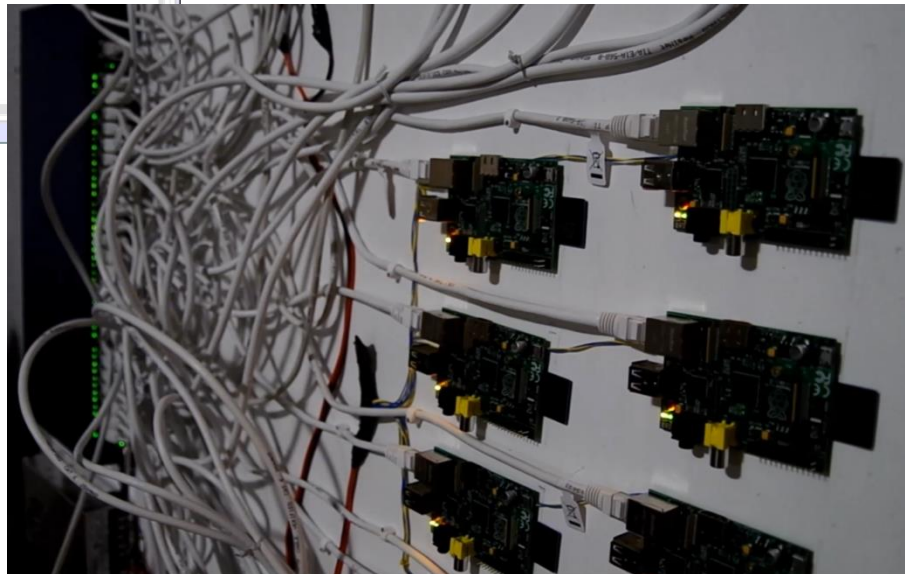
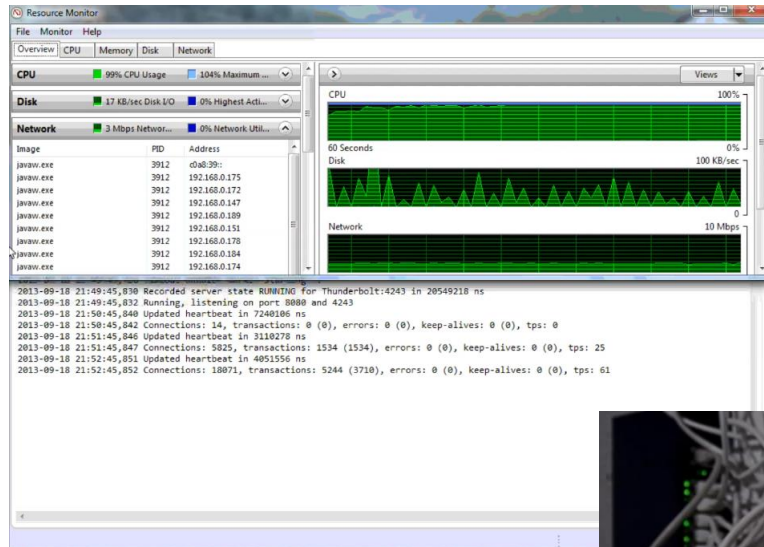
A test is only meaningful if it has clear success criteria

- 50.000 concurrent connections
- 100 business transactions per second
- Stable performance over time (at least one week)
- “Few” errors, preferably none

EXAMPLE

Load tests – Generate load...

The wall generates requests, preferably ramping up the load gradually in order not to kill the server



The times for successful tests as well as any errors should be recorded by the clients along with client side workload

- The load tests should measure and record connect times, processing times and errors – some of this can be recorded on the server as well, but only the client has the entire picture
- For complex tests the individual steps should be recorded with processing times
- Use a performance monitor on the client side – if the client is swamped (not that unlikely with a Raspberry) timeouts and other errors are likely to occur – adjust the work load accordingly!

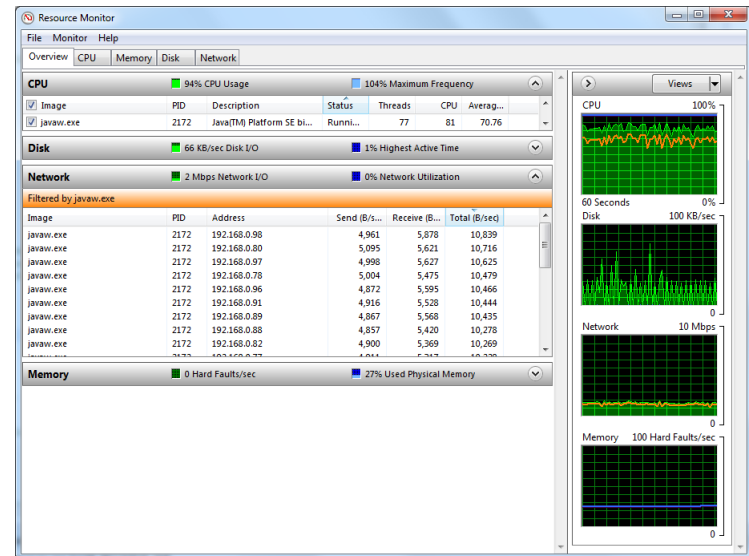
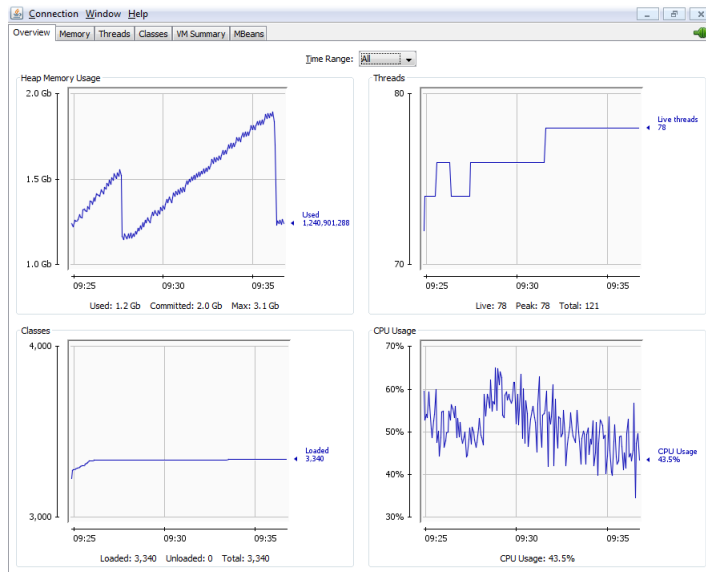
Detailed logging is expensive, it may be wise to run one pass with logging and one where it is disabled on the client side

Server-side statistics are at least as important – if possible the server should be instrumented

- Total number of successful transactions
- Total number of errors
- Number of active connections
- Number of active threads
- Processing time per step for each business transaction (if possible – this can be expensive)

These statistics can be invaluable in production











With or without instrumentation the server performance and the JVM internals should be measured



Java comes with jconsole out of the box – use it!

It is also a good idea to use a profiler in order to find bottlenecks in the code

- The visualvm profiler is included with Java SE 7 – while it is not my personal favorite, you can't beat the price and it is always there when you need it
- A profiler shows where the application spends its time – don't optimize code that never runs

Hot Spots - Method	Self time [%] ▼	Self time	Invocations
java.util.concurrent.locks.LockSupport. park (Object)		21461 ms (18.9%)	151
sun.security.ssl.Handshaker\$DelegatedTask. run ()		17630 ms (15.6%)	8541
java.util.concurrent.locks.AbstractQueuedSynchronizer. acquire (int)		12353 ms (10.9%)	76359
sun.nio.ch.WindowsSelectorImpl\$SubSelector. poll (int)		7162 ms (6.3%)	724
sun.nio.ch.WindowsSelectorImpl\$StartLock. waitForStart (sun.nio....)		3950 ms (3.5%)	729
javax.xml.parsers.SecuritySupport. getResourceAsStream (Clas...)		3626 ms (3.2%)	8620
java.util.concurrent.locks.AbstractQueuedSynchronizer. acquireQueu		1987 ms (1.8%)	491
sun.nio.ch.WindowsSelectorImpl. setWakeupSocket ()		1419 ms (1.3%)	8247
sun.nio.ch.SocketDispatcher. write (java.io.FileDescriptor, long, int)		700 ms (0.6%)	13357
java.security.MessageDigest\$Delegate. engineUpdate (byte[], int...)		540 ms (0.5%)	1036364

Note: The screenshot includes platform classes, which is unusual. With the default filters the hot spots in the application code would show up instead. Platform hot spots may also be interesting at times, in particular when comparing JVM:s.

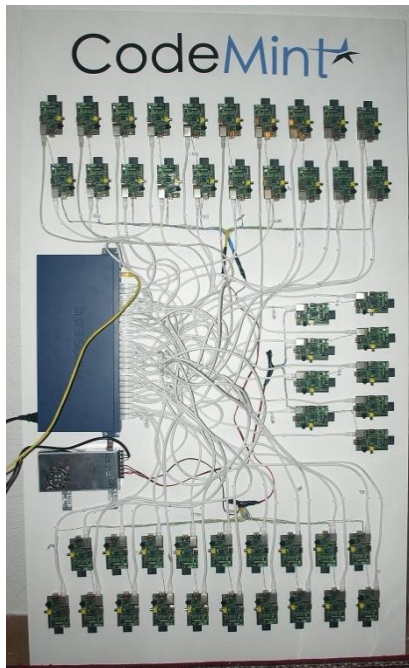
If the targets are met the test was successful – if not, the logs should indicate where the bottleneck is

- Ideally the problem stands out, perhaps in the profiler
- In more complex cases the detailed logs must be analyzed – make sure that they are in a format that can be imported into Excel and/or a database!
- Some issues are JVM or platform specific, test the application on several platforms if there are weird problems
- Google is your friend (well, at least in this case) – some issues can be very hard to track down, but chances are that someone has done it for you and blogged about it

Live demo

Depending on circumstances – test with full-size wall, Grinder demo with travel wall or video of full-scale test

- Start Tomcat
- Start Grinder console
- Start wall and agents
- Run test and measure results



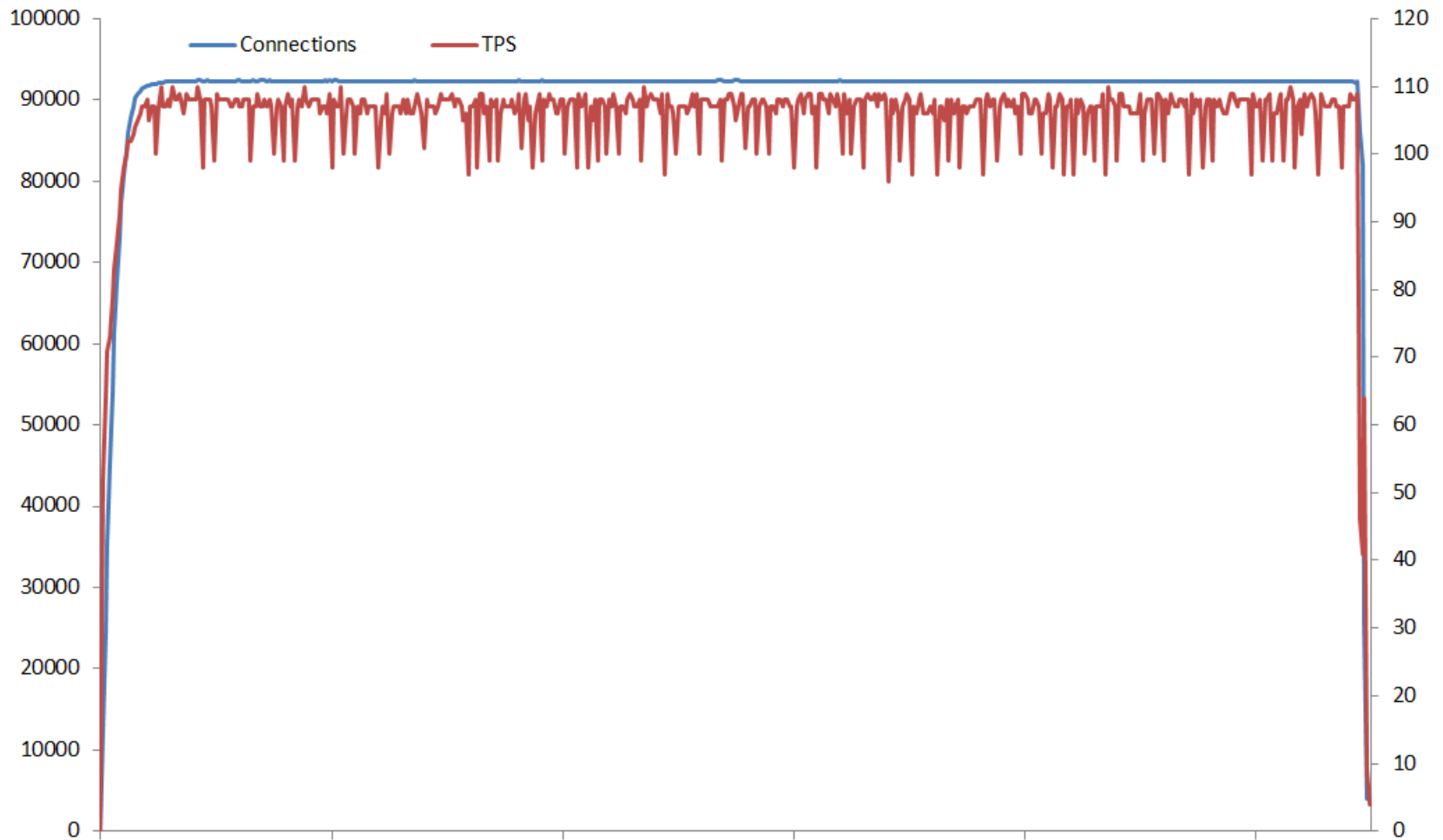
- Start custom server
- Start wall
- Start load generators
- Measure results

Results

The load tests did make a difference – we found and fixed some issues, mostly related to JVM tuning

- Profiling found some bottlenecks in the light-weight SOAP implementation that we could eliminate
- Profiling also told us what we *didn't* need to tune – SSL handshakes and XML parsing accounts for so much that improvements in our code would hardly be noticeable
- Full GC pause times caused timeout errors with the original settings – armed with that knowledge we could tune for short pause times and pay with somewhat reduced throughput
- On some platforms performance would degrade over time – we managed to work around that with JVM settings

We reached our targets and could demonstrate that the application performed as it should under extreme conditions

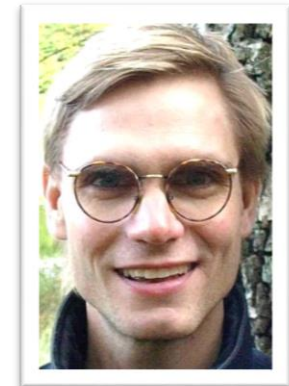


The Raspberry Wall works well as a load generator – it was rewarding to build, and we intend to put it to good use

- There have been no real issues except for a hick-up with some units that failed to boot with the soft float image and with the OpenJDK Avian crashes
- In the long term, cheap memory cards are a hazard – limit writes and invest in quality SD cards
- The normal Java load test frameworks can be used, indeed our existing Grinder tests worked without any changes
- The platform seems to get better all the time – updates are released and there is a wealth of documentation on the Internet
- We are very happy that Oracle has released their JVM for Raspbian and look forward to the official Java 8 release!



Questions?



Erik Wramner
CodeMint AB
Mässans gata 18
412 51 Göteborg
Sweden
<http://codemint.com>

