Αρχική σελίδα

Από Gaia Wiki

Πίνακας περιεχομένων

Error reporting

A Mantis ticket handling (for internal use by GAIA developers) resides here:

```
GAIA ticket handling (http://172.31.128.119/mantisGAIA)
```

Coding aspects

Checking out the source code

The subversion repository for the preliminary code resides here:

```
https://172.31.128.119/svn/neuro-jsf-pilot/trunk
```

To checkout, simply:

```
$ svn co --username=<YOURUSERNAME> https://172.31.128.119/svn/neuro-jsf-pilot/trunk GAIA
```

Building with Ant

First, you'll need JDK 1.7 installed. Then:

```
$ cd GAIA/neuro-jsf-pilot-jacket/neuro-jsf-pilot
$ mkdir $HOME/jboss-standalone-deployments
$ ant
```

The build process will create `Orders.ear` in the `$HOME/jboss-standalone-deployments` folder.

Install JBoss and after the build completes, copy the `Orders.ear` file there. If all goes well, you should be able to see GAIA in action - via:

```
http://<yourmachineip>:8888/orders
```

Building with NetBeans

If you prefer using NetBeans to build, then after checking out, you need to add this line (appropriately modified to point to the folder where you checked out)

```
var.neuro_pilot_jacket=/home/ttsiod/work/neuro-jsf-pilot/neuro-jsf-pilot-jacket
```

...into this file:

```
~/.netbeans/7.1.2/build.properties
```

Then start NetBeans, and open the GAIA/neuro-jsf-pilot-jacket/neuro-jsf-pilot-NB project - making sure that you have checked the checkbox "Open Required Projects".

You should be able to build after this - but deployment must still be done via JBoss (so copy the generated `Orders.ear` to JBoss's deployment folder.

Registering Java 7 in the NetBeans IDE

```
1. Download and install JDK 7 on your system.
2. In the IDE, choose Tools > Java Platforms from the main menu.
3. Click Add Platform and specify the directory that contains the JDK (e.g. on Windows, this is the JDK instal
4. In the Platform Name step, verify that the default locations of the Platform Sources zip file and API docum
5. Click Finish to close the Add Java Platform dialog box.
```

You must also configure glassfish to start with Java 7

```
1. Go to Services tab on your IDE.
2. Expand Servers tree.
3. Choose Properties on your Glassfish server.
4. Choose java tab and set Java Executable to your java.exe file of JDK 7 installation.
```

Administration aspects

Installing JBoss in cluster mode

For the gaia-devapp1 (172.31.128.103) and gaia-devapp2 (172.31.128.104) machines, I followed the instructions from the official JBoss site (https://docs.jboss.org/author/display/AS71/AS7+Cluster+Howto?_sscc=t) .

Important: since the two machines were cloned with the exact same JBoss installed, the cluster came up with an error about two nodes with the same node-id being alive at the same time. To cope with that, I had to clear up the server.lock files that dictate HornetQ when to create a new node-id:

```
 # cd /opt/jboss/jboss-as-7.1.3.CR1/domain
 # find . | grep server.lock
 ./domain/servers/server-two/data/messagingjournal/server.lock
 ./domain/servers/server-one/data/messagingjournal/server.lock
 ./domain/servers/server-three/data/messagingjournal/server.lock

 # rm $(!!)
```

Update, 2012/10/23: This magic option must be added to JBoss run script (standalone.sh or domain.sh)

```
 -Dorg.apache.el.parser.COERCE_TO_ZERO=false
```

It makes form fields get "null" value when the control is empty (instead of the stupid casts JBoss does (Integer=0, Boolean=false, etc)

Monitoring of Load Balancers

Apache Pacemaker Cluster Monitor

```
http://172.31.128.113/
http://172.31.128.114/
```

Apache mod_cluster Load Balancer Monitor

```
http://172.31.128.113:10001/mod_cluster-manager
http://172.31.128.114:10001/mod_cluster-manager
```

Enterprise DB

Clustered EnterpriseDB in failover mode on nodes 172.31.128.106 and 172.31.128.107

ClusterDB IP: 172.31.128.116

Cluster status:

```
http://172.31.128.106:2223
http://172.31.128.107:2223
```

Installing PostgreSQL JDBC Driver in JBoss

Φτιάχνουμε το directory tree $jboss_home/modules/org/postgresql/main

Κατεβάζουμε στο παραπάνω directory τον PostgreSQL JDBC Driver http://jdbc.postgresql.org/

Στον ίδιο φάκελο φτίαχνουμε το αρχείο module.xml με τα παρακάτω περιεχόμενα

<module xmlns="urn:jboss:module:1.1" name="org.postgresql">

```
<resources>
  <resource-root path="postgresql-9.1-902.jdbc4.jar"/>
</resources>
<dependencies>
  <module name="javax.api"/>
  <module name="javax.transaction.api"/>
</dependencies>
```

</module>

(Στο resource root βάζουμε το όνομα του jar που έχουμε κατεβάσει)

Κάνουμε register τον driver μεσω του jboss-cli

-Σε managed domain εγκατάσταση οπως λειτουργεί τώρα:

από το directory $jboss_home/bin τρέχουμε

./jboss-cli.sh --connect controller=<ip του master server>:9999 command="/profile=full-ha/subsystem=datasources/jdbc-driver=postgresql-driver:add(driver-name=postgresql-driver, driver-class-name=org.postgresql.Driver, driver-module-name=org.postgresql)"

-Σε standalone εγκατάσταση

από το directory $jboss_home/bin τρέχουμε

./jboss-cli.sh --connect command="/subsystem=datasources/jdbc-driver=postgresql-driver:add(driver-name=postgresql-driver, driver-class-name=org.postgresql.Driver, driver-module-name=org.postgresql)"

Installing EnterpriseDB JDBC Driver in JBoss

Φτιάχνουμε το directory tree $jboss_home/modules/org/postgresql/main

Κατεβάζουμε στο παραπάνω directory τον PostgreSQL JDBC Driver [\\172.31.128.250 \Download\Setup Programs\PostgreSQL\JDBC Drivers\EnterpriseDB \\172.31.128.250 \Download\Setup Programs\PostgreSQL\JDBC Drivers\EnterpriseDB]

Στον ίδιο φάκελο φτίαχνουμε το αρχείο module.xml με τα παρακάτω περιεχόμενα

<module xmlns="urn:jboss:module:1.1" name="com.edb">

```
<resources>
  <resource-root path="edb-jdbc16.jar"/>
</resources>
<dependencies>
  <module name="javax.api"/>
  <module name="javax.transaction.api"/>
</dependencies>
```

</module>

(Στο resource root βάζουμε το όνομα του jar που έχουμε κατεβάσει)

Κάνουμε register τον driver μεσω του jboss-cli

-Σε managed domain εγκατάσταση οπως λειτουργεί τώρα:

από το directory $jboss_home/bin τρέχουμε

./jboss-cli.sh --connect controller=172.31.128.103:9999 command="/profile=full-ha/subsystem=datasources/jdbc-driver=edb-driver:add(driver-name=edb-driver, driver-class-name=com.edb.Driver, driver-module-name=com.edb)"

-Σε standalone εγκατάσταση

από το directory $jboss_home/bin τρέχουμε

./jboss-cli.sh --connect command="/subsystem=datasources/jdbc-driver=edb-driver:add(driver-name=edb-driver, driver-class-name=com.edb.Driver, driver-module-name=com.edb)"

JBoss Application Servers Monit Process Manager

Η εκκίνηση-τερματισμός των JBoss servers γίνεται μέσω του Monit στα urls:

```
http://172.31.128.103:2222
http://172.31.128.104:2222
```

Configuration of JBoss datasource for EnterpriseDB

Το hibernate δεν καταλαβαίνει αυτόματα την διάλεκτο EnterpriseDB, και πρέπει να του το πούμε explicitely στο .../webapp/APPNAME/cashflow-ejb/src/conf/persistence.xml :

```
...
<properties>
    <property name="hibernate.dialect" value="org.hibernate.dialect.PostgresPlusDialect"/>
</properties>
...
```

Επιπλέον, το datasource πρέπει να χρησιμοποιεί τον edb-driver, και το JDBC string να είναι της μορφής:

```
jdbc:edb://172.31.128.116:5444/APPNAME
```

Testing aspects

Stress testing the Enterprise DB

We created a simple schema, containing a master and a detail table. Three scripts were written for each of the two tables - they are part of the repository, under folder "testing/StressTests/DB.cluster":

```
bash$ cd ~/GAIA/testing/StressTests/DB.cluster/

bash$ ls -l
total 36
drwxr-xr-x 3 ttsiod ttsiod 4096 Νοέ   2 15:55 masterDetailTables
drwxr-xr-x 3 ttsiod ttsiod 4096 Νοέ   2 14:25 standaloneTable
-rw-r--r-- 1 ttsiod ttsiod 9050 Νοέ   2 13:49 stressTest.py

bash$ cd standaloneTable/

bash$ ls -l *py
-rwxr-xr-x 1 ttsiod ttsiod 2699 Νοέ   2 14:24 createTableOracle.py
```

```
-rwxr-xr-x 1 ttsiod ttsiod 2256 Οκτ  26 18:41 createTablePostgres.py
-rwxr-xr-x 1 ttsiod ttsiod  424 Νοέ   2 12:57 insertTable.py
-rwxr-xr-x 1 ttsiod ttsiod  356 Νοέ   2 12:57 searchTable.y

bash$ cd ../masterDetailTables

bash$ ls -l *py
-rwxr-xr-x 1 ttsiod ttsiod 2098 Νοέ   2 15:55 createDetailTableOracle.py
-rwxr-xr-x 1 ttsiod ttsiod 2379 Νοέ   2 15:55 createDetailTablePostgres.py
-rwxr-xr-x 1 ttsiod ttsiod  704 Νοέ   2 15:55 insertDetailTable.py
-rwxr-xr-x 1 ttsiod ttsiod  362 Νοέ   2 15:55 searchDetailTable.py
```

The scripts depend on a properly set ~/.pgpass file - with e.g. contents like:

```
172.31.128.116:5444:stress:gaiauser:THEPASSWORD
172.31.128.116:5444:stress:enterprisedb:THEOTHERPASSWORD
```

Data Creation

To do an indicative benchmark, the two *create...Table.py* scripts:

- open a connection to the database
- open a cursor under that connection
- and invoke a series of create statements

In order to compare with Oracle, the *create...* scripts exist for Oracle as well. The two tables are created with the following attributes:

- The *stressTest* table contains:
  - an *id* column, of type SERIAL (primary key)
  - a *name* column, of type VARCHAR(30)
  - a *price* column of type NUMBER(8,2)
  - 90 columns with random names of type VARCHAR(40)
  - 10 columns with random names of type date

The *name* and *price* columns are indexed.

Note: For Oracle, since there is no SERIAL type, we created the corresponding sequence and a trigger that uses it upon insertion.

Bulk load and full table scans

We tried the 'COPY' command from within postgres - **which imported 1M records in 70 seconds, achieving a rate of around 15K records inserted per sec.**. The input file data size was around 1GB.

Once the data were loaded in the DB, we did a couple of simple queries to see how Postgres performs on full table scan queries:

```
$ time sh -c "echo 'select count(distinct(name)) from stresstest' | psql -U gaiauser -h 172.31.128.116 -p 544
  count
---------
```

```
 1000000
real    0m5.968s
user    0m0.052s
sys     0m0.012s
```

```
$ time sh -c "echo 'select sum(price) from stresstest' | psql -U gaiauser -h 172.31.128.116 -p 5444 stress"
     sum
--------------
 500678089.34
real    0m0.915s
user    0m0.032s
sys     0m0.024s
```

DB benchmarking tool

To further test normal insertions and searches, we developed a custom tool that we used to benchmark both Oracle and PostgreSQL. The tool allows the user to write any query he wants, and see it executed by a number of processes spawning a number of threads.

Here is the usage screen of the tool we created:

```
Neuropublic DB benchmarker, revision: $Rev: 677 $
Usage: ... <options>
Where options are:

-h, --help              Show help
-v, --version           Show version number

Common for all database engines:

-c, --processes <N>     How many processes to use (default: 15)
-t, --threads <N>       How many threads to use   (default: 10)
-d, --duration <N>      How many seconds to run   (default: 5)
-u, --user <username>   Database username         (defaults:
                         for Oracle:stresstest, for PostgreSQL:gaiauser)
-p, --password <pass>   Database password         (defaults:
                         for Oracle:stresstest, for PostgreSQL:gaia-user-pwd)

PostgreSQL specific:

-1, --postgres          Benchmark with Postgres
-s, --host <address>    Postgres IP address   (default:172.31.128.116)
-r, --port <port>       Postgres port         (default: 5444)
-b, --db <dbName>       Postgres databaseName (default: stress)

Oracle specific:

-2, --oracle            Benchmark with Oracle
-n, --dsn <dsn>         Oracle dsn (default:172.31.128.52:1521/orcl)
```

Benchmarking template

To create a new benchmark using the tool, you follow this simple Python template:

```
#!/usr/bin/env python

import sys
import random

```

```
sys.path.append("..")
from stressTest import Benchmark, getRandomName


def doWork(cur, unused):
    cur.execute("insert into stressTest(name, price) values ('%s', %d.%d)" %
                (getRandomName(10), random.randint(0, 1000), random.randint(0, 99)))


def setup(cur):
    return None


def main():
    Benchmark(setup, doWork)


if __name__ == "__main__":
    main()
```

Any benchmarking script, simply imports the *Benchmark* function from the *stressTest* library, and invokes it, passing two functions:

- the setup function is supposed to do any kind of work that needs to be done *before* spawning the processes. Whatever it returns, will be passed on - unchanged - as the second parameter into doWork.
- the doWork function is the one that will run all the time. Basically, the Benchmark function will spawn a number of processes, and each process will spawn a number of threads - each thread will run a continuous loop, that invokes the *doWork* function.

So, looking at the code above, we see a script that will insert random values into the *name* and *price* columns of the *stressTest* table.

We wrote similar code for the search (i.e. select) script, and did it again for the detail table. Note that in the detail table, we needed to learn the range of available *id*s in the master table - so the setup function does this:

```
def doWork(cur, maxId):
    cur.execute("insert into stressTestDetail(master_id, name, price) "
                + " values (%d, '%s', %d.%d)" % (random.randint(1, maxId),
                                                 getRandomName(10),
                                                 random.randint(0, 1000),
                                                 random.randint(0, 99)))


def setup(cur):
    cur.execute("select max(id) from stressTest")
    maxId = cur.fetchone()[0]
    return maxId
```

In plain words, we first learn the maximum id available in the master table, and then, in the doWork function executed by each thread, we insert a random foreign key value selected inside that range.

At the end, each script automatically reports MIN/AVG/MAX time taken by each query it executed.

Let's see the results we obtained:

Comparisons with Oracle

Insertions in the master table

We started from a fresh, clean DB - right after a *./createTablePostgres.py* invocation.

We then executed this code in the doWork function:

```
cur.execute("insert into stressTest(name, price) values ('%s', %d.%d)" %
            (getRandomName(10), random.randint(0, 1000), random.randint(0, 99)))
```

...and using 15 processes, with 20 threads per process, we got these results from a 30 sec execution in our Postgres devDB cluster (172.31.128.116):

```
bash$ ./createTablePostgres.py

bash$ ./insertTable.py --postgres -c 15 -t 20 -d 30

Beginning benchmark of PostgreSQL
Number of processes: 15
Number of threads: 20
Number of seconds: 30
Benchmark completed.

Minimum :
  Average value: 0.00191
  Std deviation: 0.00026
  Sample stddev: 0.00026
         Median: 0.00192
            Min: 0.00123
            Max: 0.00268
   Overall: 0.00190766255061 +/- 13.9%

Average :
  Average value: 0.05986
  Std deviation: 0.00222
  Sample stddev: 0.00223
         Median: 0.05970
            Min: 0.05434
            Max: 0.06577
   Overall: 0.0598633026432 +/- 3.7%

Maximum :
  Average value: 0.27941
  Std deviation: 0.04359
  Sample stddev: 0.04367
         Median: 0.26931
            Min: 0.19696
            Max: 0.48286
   Overall: 0.279405450026 +/- 15.6%

  Total queries: 150814
           Rate: 5001.61
```

The script reports statistics on the minimum, average and maximum times taken to execute the doWork function, per thread. One can see that **the longest insert took half a second, but the average insert took 60ms**.

Looking at the final line, we note that **PostgreSQL can insert around 5K records per second, in an empty table, where only 2 columns actually get data** (3 if we include the *id*).

We then did the same test in Oracle (172.31.128.52):

```
bash$ ./createTableOracle.py

bash$ ./insertTable.py --oracle -c 15 -t 20 -d 30

Beginning benchmark of Oracle
Number of processes: 15
Number of threads: 20
Number of seconds: 30
Benchmark completed.

Minimum :
  Average value: 0.10932
  Std deviation: 0.02727
  Sample stddev: 0.02731
         Median: 0.11134
            Min: 0.02240
            Max: 0.17335
   Overall: 0.109321631591 +/- 25.0%

Average :
  Average value: 0.37074
  Std deviation: 0.01664
  Sample stddev: 0.01667
         Median: 0.36864
            Min: 0.33015
            Max: 0.43155
   Overall: 0.370739942335 +/- 4.5%

Maximum :
  Average value: 0.92818
  Std deviation: 0.12313
  Sample stddev: 0.12334
         Median: 0.91533
            Min: 0.69410
            Max: 1.33230
   Overall: 0.92817956686 +/- 13.3%

  Total queries: 24481
           Rate:  796.04
```

The results are indeed inferior - but note that we are using different servers, so the difference may very well exist because of the difference in their HW.

It is also of note that **the benchmark was inserting in an empty table** - the behaviour as the table gets more and more data is also of interest. Here is what happens to the performance when we have already inserted 1200000 records (1.2M):

```
bash$ psql -U gaiauser -h 172.31.128.116 -p 5444 stress
psql (9.1.6)
Type "help" for help.

stress=> select count(*) from stressTest;
  count
---------
 1229063
(1 row)
```

```
stress=> \q

bash$ ./insertTable.py --postgres -c 15 -t 20 -d 30

Beginning benchmark of PostgreSQL

...
Average :
   ...
   Overall: 0.0645483690019 +/- 4.3%
   ...
   Total queries: 139966
           Rate: 4640.17
```

Performance **dropped by 7%** compared to an empty table.

Oracle however, remained at the same level of performance:

```
bash$ sqlplus stresstest/stresstest@172.31.128.52:1521/orcl

SQL*Plus: Release 11.2.0.3.0 Production on Fri Nov 2 17:30:46 2012

Copyright (c) 1982, 2011, Oracle.  All rights reserved.

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 64bit Production
With the Partitioning, Automatic Storage Management, OLAP, Data Mining
and Real Application Testing options

SQL> select count(*) from stressTest;

  COUNT(*)
----------
   1120052

SQL> Disconnected from Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 64bit Production
With the Partitioning, Automatic Storage Management, OLAP, Data Mining
and Real Application Testing options

bash$ ./insertTable.py --oracle -c 15 -t 20 -d 30
...
Average :
   ...
   Overall: 0.37144260521 +/- 3.9%
   ...
   Total queries: 24382
           Rate:  799.15
```

Searches in the master table

For search (i.e. select queries), we used this *doWork* block:

```
searchName = getRandomName(10)
cur.execute("select * from stressTest WHERE name='%s'" % searchName)
```

...and got these results in 5-sec execution in Postgres (172.31.128.116):

```
Average : 0.00342174099374 +/- 4.4%
Maximum : 0.0208919843038 +/- 36.7%
Total queries: 21905
Rate: 4326.77 (/sec)
```

...and these in 5-sec execution in Oracle (172.31.128.52):

```
Average : 0.0157786937801 +/- 1.7%
Maximum : 0.059529097875 +/- 18.1%
Total queries: 4757
Rate:  917.51 (/sec)
```

Insertions in the detail table

PostgreSQL detail table insertions:

```
bash$ ./insertDetailTable.py  --postgres -c 15 -t 20 -d 30
...
Average : 0.0708479645294 +/- 3.3%
Maximum : 0.341773080031 +/- 18.1%
...
  Total queries: 127443
           Rate: 4223.04
```

Oracle detail table insertions:

```
bash$ ./insertDetailTable.py  --oracle -c 15 -t 20 -d 30
...
Average : 0.358633601487 +/- 4.1%
Maximum : 1.0106039532 +/- 20.3%
...
  Total queries: 25302
           Rate:  829.12
```

Searches in the detail table

PostgreSQL detail table searches:

```
bash$ ./searchDetailTable.py  --postgres -c 15 -t 20 -d 30
...
Average : 0.0591614059637 +/- 6.0%
Maximum : 0.251888020833 +/- 15.8%
...
  Total queries: 152916
           Rate: 5074.26
```

Oracle detail table searches:

```
bash$ ./searchDetailTable.py  --oracle -c 15 -t 20 -d 30
...
Average : 0.459356089685 +/- 3.9%
Maximum : 0.914564186732 +/- 12.6%
...
  Total queries: 19749
           Rate:  644.91
```

How full is the DB server?

We used a nested shell loop to invoke the benchmark over the span of 1-20 processes, and 1-20 threads. This is what we got:

Preliminary benchmarking results

These results allow for cautious optimism regarding PostgreSQL.

Note however that the results came from different machines, so they are not directly comparable.

All they are telling us, is that our current Postgres EnterpriseDB installation doesn't appear to have any obvious bottlenecks.

Stress testing JBoss EAP

Burst-testing of session-baskets

Using the session-baskets code (https://github.com/mperdikeas/archetypes/tree/master/jsf/007-01-session_basket-clustered-B_ttsiod_stresstesting) , which stores session data in memory (i.e. it doesn't use the DB), we can test the failover behaviour of EAP. We've seen in the instructions above, how the JBoss EAP has been configured as a a master-slave cluster - with failover enabled.

To proceed with the test, we added test code in the repository (https://172.31.128.119/svn/neuro-jsf-pilot/trunk/testing/cluster/test-cluster-deployment-of-session-baskets-on-EAP.py) , which makes use of *Twill* to automate the "clicks" a user would do on the application - and thus allows bursts of user activity, far faster than a human ever could.

Starting from the session basket main page (http://172.31.128.113/session-basket/) , the test script:

- Hits the "create" button

...then runs this loop, 500 times:

- Enters a random item
- Clicks on "save"
- verifies the page responds with the added element

When the loop ends...

- it returns to the main page - which shows the entries list
- The code then verifies that *ALL* 500 entries are there.

This allows easy verification of the EAP behaviour

Failover behaviour: Setup

We added two bash shortcuts to (a) *list* the PIDs of Java JBoss processes on server 1 or

2, and (b) *stop* to kill these PIDs in server 1 or 2:

```
$ tail .bashrc

function list()
{
    ssh gaia-devapp"$1" "ps aux | grep 'java.*b[o]ss' | awk '{print \$2}'"
}

function stop()
{
    ssh gaia-devapp"$1" "ps aux | grep 'java.*b[o]ss' | awk '{print \$2}' | while read ANS ; do kill -9 \$ANS
}
```

Using SSH keys, we skipped past password authentication - and were able to invoke these one-liners, to list or kill all JBoss app instances in either devapp1 or devapp2:

```
ttsiod@avalon ~/GAIA/testing/cluster
$ list 1
1470
1486
1550
1569
1578
```

The listed PIDs of the JBoss java instances, can then be stopped with:

```
ttsiod@avalon ~/GAIA/testing/cluster
$ stop 1
```

```
ttsiod@avalon ~/GAIA/testing/cluster
$ list 1
```

This is how we investigated how the EAP failover operates - we spawned our test script, and stop-ed either devapp1 or devapp2 while it was running. Monit was configured to automatically restart them after 10-15 seconds.

Failover behaviour: Using NGINX

With this "round-robin" configuration...

```
upstream myproject {
        server 172.31.128.103:8330;
        server 172.31.128.104:8330;
}
server {
        listen 80;
        server_name 172.31.128.113;
        location / {
                proxy_pass http://myproject;
        }
}
```

What we saw was that requests from the same session were dispatched in round-robin fashion to both servers. We tested stopping one server while the script was running, and saw that the requests were transparently forwarded to the remaining server - i.e. the "user" didn't understand anything about the problem. However, when we (a) waited for server 1 to be respawned by Monit, and then (b) proceeded to kill server 2 (hoping that user's requests would bounce back to server 1)... we saw the script reporting 404s.

Executive summary: with NGINX, we have session-failover for the first down server - but even if we wait for the 1st server to recover, there is no 2nd session failover when we kill the 2nd one.

Failover behaviour: Using HA-proxy

The original HA-proxy configuration used in Neuropublic, was using the session cookies to pick one server at the beginning of the session - and would then "stick" to it for the duration of the session. What this means, was that when we killed the server that was serving the session, the user would instantly see 404s - there was no session failover at all.

We then found out how to configure HA-proxy to behave just as NGINX did - i.e. with "dumb" round-robin. We then witnessed the exact same behaviour as we did with NGINX - i.e. we have session-failover for the first down server - but even if we wait for the 1st server to recover, there is no 2nd session failover when we kill the 2nd one.

d_koukoutsis: document what you changed in the HA proxy to achieve this

Failover behaviour: Using mod_cluster

The best failover behaviour was witnessed when we used the custom Apache mod_cluster module, which gave us session failover even for the 2nd kill - i.e. this scenario worked:

1. we spawn the script
2. we see it's requests being delivered to devapp1
3. from another window, we stop devapp1
4. we see the requests now being delivered to devapp2 (i.e. the user sees no problem)
5. we wait for monit to respawn devapp1
6. when devapp1 is up, we stop devapp2
7. we see the requests now being delivered back to devapp1 (i.e. the user again sees no problem)

Monitoring for mod_cluster is available at:

http://172.31.128.113:10001/mod_cluster-manager

PostgreSql

Management Queries

Active Sessions

```
SELECT * FROM pg_stat_activity;
```