A Private PaaS with Mongrel2 and ZeroMQ

Loïc d'Anterroches for Céondo Ltd

Fri 27 Jan, 2012

PaaS or Platform as a Service, you cannot sell or advertise a piece of software at the moment without packaging it into some kind of PaaS friendly packaging. Today I will show you how to create your own PaaS with Mongrel2, Git (or your VCS of choice) and ZeroMQ.

The requirements are pretty basic, you just need a couple of servers with both a public and a private address. Our PaaS is running with Ganeti, but you can use Amazon EC2.

Mongrel2, a Langage Agnostic and Backend Friendly Webserver

What is Mongrel2?

> Mongrel2 is an application, language, and network architecture agnostic web server that focuses on web applications using modern browser technologies.

The ground breaking features are for me:

- the support of many protocols over the same port 80/443. For example you can keep a WebSocket connection open at the same time you serve HTTP 1.1 on port 80. You do not need a second server over another port (which could be firewalled) to manage this extra protocol;
- all the requests are parsed and sent to the backend as simple to handle ZMQ messages;
- everything is asynchronous and thus you *address* an answer to a *given* client or set of clients. Just this is mind blowing when you do realtime applications.

ZeroMQ, the new Lingua Franca in the Backend

ZeroMQ is a library which provides you something looking like traditional Unix sockets but totally awesome to deliver messages between processes (or even threads in a process) simply and efficiently. You can create a server in 6 lines of Python answering 1000's of requests per second:

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

import zmq
import time

context = zmq.Context()
socket = context.socket(zmq.REP)
socket.bind("tcp://*:5555")

while True:
    #  Wait for next request from client
    message = socket.recv()
    print "Received request: ", message

    # Do some 'work', of course you will not
    # serve 1000's of requests per seconds with this!
    time.sleep(1)

    #  Send reply back to client
    socket.send("World")
```

With hundreds of examples, the ZeroMQ guide is a must read.

Connecting the Dots, What is a PaaS?

A Platform as a Service is a bit like shared hosting — where 1000's of websites share the same infrastructure — with a bit more constraints and defined deployment procedures to provide:

1. a simple way to create, update web applications;
2. a systematic way to deploy, scale up and down the applications.

The enforced constraints, for example the fact that you are running your application on a read only file system, are the key to allow the last point. You lose a little bit of flexibility to gain peace of mind in your day to day work.

The PaaS Workflow

The traditional PaaS workflow — pioneered by Heroku and know used by a dozen if not more PaaS providers — links the *creation* part and the *deployment* part with the version control tool. The idea is that when you send the provider a new version of your code, it automatically uses it to create a new version of your application, deploy it, stop the old version and start the new one.

At Céondo, we have our own PaaS named Bareku as an hommage to Heroku. So here is a typical workflow I have when developing Cheméo:

1. new idea or bug to fix;
2. write the code, run the unit tests and play with it locally. I create a branch or not, but usually I commit many times locally to keep track of my work;
3. when satisfied, I push the code to my *Bareku* repository: `$ git push bareku master`;
4. in the pre-receive hook of Git, the *latest master* is checked out and verified. The application is packaged and deployed. If this is successful, the code is accepted.

What is really nice is that as a coder, you basically deploy a new version with just a single command. For example for Cheméo:

```
$ echo "This README is not up-to-date" >> README
$ git commit -a -m "Informed that the README is not up-to-date."
[master e76d105] Informed that the README is not up-to-date.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git push origin master
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 360 bytes, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: -----> chemeo app.
remote: -----> PHP/Photon project detected.
remote:        Running with php-5.3.8.
remote: -----> Fetching latest Photon.
remote: -----> Building the .phar project file.
remote: -----> Building the slug file.
remote:        Compiled slug size is 488KB.
remote:        Build: 2765e91b-50c4-4d04-b92c-f38f82661b2a
remote: -----> Deploying to the nodes.
remote:        192.168.1.125 - OK.
remote:        192.168.1.111 - OK.
remote: -----> Set application runtime configuration.
```

```
remote:        Application version: 41
remote:        Scale web to 3 processes.
remote: -----> Job done.
To git@git.bareku.com:repositories/chemeo.git
   9c07751..e76d105  master -> master
```

Now, take a look at the homepage of Chemeo, you can see `v41` at the bottom. So, this means that within 5 seconds, without leaving my development tools, I created a new version of the application, tracked the code to be able to possibly revert/amend my work and deployed it in production. Time to learn how to do it and create your own PaaS.

Bareku, the Mongrel2 and ZeroMQ PaaS

Bareku is heavily inspired by Heroku but built as a *private* PaaS, that is, it is built to run trusted applications for our company. This relaxes a lot the deployment constraints as we do not have to manage a kind of firewall between the applications and this allow us to create different applications communicating easily together. In our case this is critical to interconnect web and scientific applications.

A clean setup requires at least 3 servers or virtual machines — named *nodes* in the rest of this note — as shown on the following diagram:

- the application repository hosting — Repository Node;
- the nodess running the applications — App Node;
- the front-end running Mongrel2 — Frontend Node.

The service nodes, providing things like PostgreSQL or MongoDB are managed independently. The coders push code with Git to the repository node, where the new application is built and then deployed to the application nodes.

Core Bareku Principles

These are the principles and assumptions when building this PaaS, the order is not really relevant:

- **fun**, it must be fun and painless to use it.
- **minimal coupling**, if a component goes down, it stops providing its services but not bring down others in a cascade.
- **trusted applications**, each application can easily access other applications over ZeroMQ, critical to give webapps access to scientific apps.

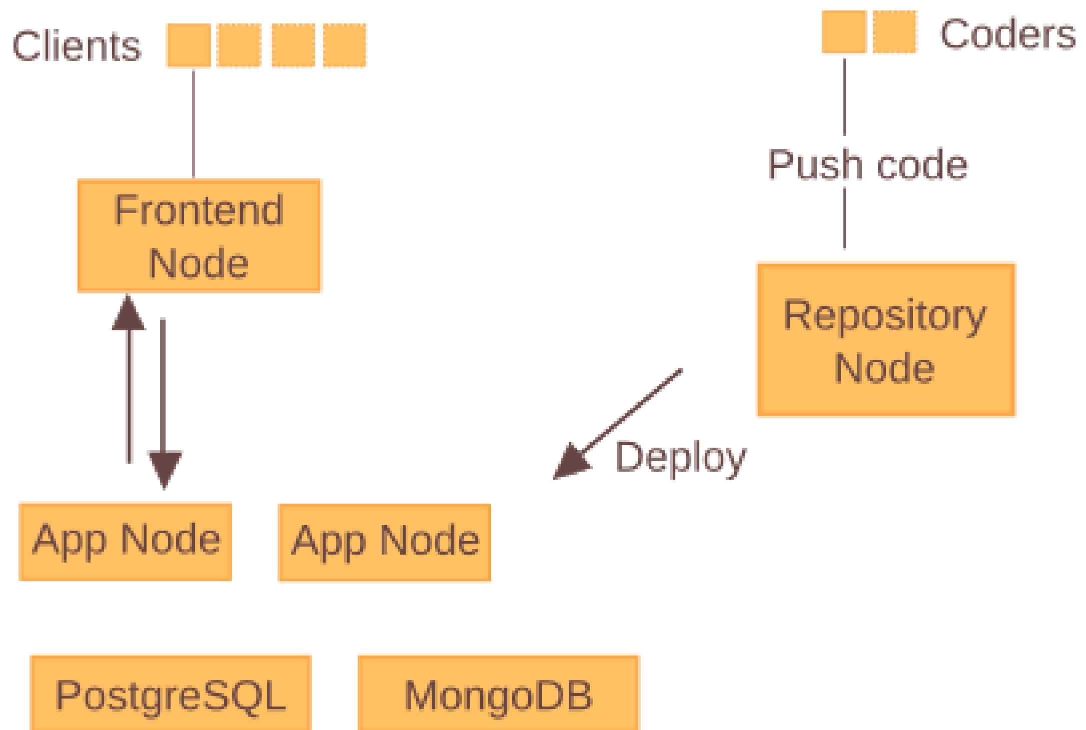

Figure 1: The simple architecture of the Paas

- **scalable**, the design must allow easy scaling to support the load while staying simple.
- **Zero downtime update**, update of an application must result in zero downtime.
- **Secure**, applications are running under the `nobody` user, special SSH key pairs for internal distribution, logging.

git.bareku.com the Application Repository Hosting

The application repository hosting node is the workhorse of the system. It takes care of:

- hosting the source code;
- building the new version in the pre-receive Git hook;
- keeping track of all the application states (version, environment, etc.);

- pushing the new version on the application nodes;
- creating the new diprocd master configuration.

But, the setup is really simple:

- one Git repository per application;
- a pre-receive Git hook;
- and the users authenticated with SSH key pairs.

pre-receive Hook

Take a look at the Heroku buildpacks, you will find for each one some files like `detect`, `compile`, etc. They are executed by **Heroku** in the `pre-receive` hook. Here is the hook we have:

```
#!/usr/bin/env bash
# bin/pre-receive
set -e
FULL_GIT_DIR=$(cd "$GIT_DIR" && /bin/pwd || "$GIT_DIR")
BIN_DIR=$(dirname $(readlink $0))
SCRATCH_DIR=/home/bareku/scratch
CACHE_DIR=/home/bareku/cache
DEF_DIR=/var/lib/bareku/apps
GIT_DIR_NAME=$(basename $FULL_GIT_DIR)
GIT_DIR_NAME=${GIT_DIR_NAME%.*}
echo "-----> $GIT_DIR_NAME app."
while read OLDREV NEWREV REF
do
    if [ "$REF" != "refs/heads/master" ]; then
        echo "$REF is not master"
        continue
    fi
    rm -rf $SCRATCH_DIR/$GIT_DIR_NAME
    mkdir -p $SCRATCH_DIR/$GIT_DIR_NAME $CACHE_DIR/$GIT_DIR_NAME
    GIT_WORK_TREE=$SCRATCH_DIR/$GIT_DIR_NAME git checkout -f $NEWREV &> /dev/null
    $BIN_DIR/compile $SCRATCH_DIR/$GIT_DIR_NAME $CACHE_DIR/$GIT_DIR_NAME $GIT_DIR_NAME
    VER=$(cat $DEF_DIR/$GIT_DIR_NAME.ver)
    echo "$VER $NEWREV" >> $DEF_DIR/$GIT_DIR_NAME.versions
done
# Very important to return 0 for the new revision to be accepted
exit 0
```

What is important for you to notice is that:

- we work only on the `master` branch;
- we keep track of some states in some `bareku` folders (which need to be writable by the git user);
- we `compile` the application with our Heroku inspired buildpack.

If you take a look at our buildpack, the additions compared to Heroku's are:

- `makeconf.py`: to generate the runtime configuration;
- `deploy`: to push the new version on the application nodes.

Is it robust? With around 10 applications running at the moment (both Python and PHP/Photon applications) and more than 200 deployments, I have noticed only one thing: **I code faster and better because I removed friction in the process**.

App Nodes, the Application Runtime Nodes

The runtime nodes are simple because the applications themselves are nearly self contained. The Python applications use `virtualenv` and the PHP/Photon applications use a defined version of PHP with phpfarm.

So, what is really on these nodes?

```
$ ls /home/slugs/
00191ebe-d349-413c-9f3a-5a1988d5648d
00191ebe-d349-413c-9f3a-5a1988d5648d.sqsh
035eaf8e-88c2-497a-9179-1447a3892225
035eaf8e-88c2-497a-9179-1447a3892225.sqsh
```

Yes, each version is transferred from the build/hosting node to the application nodes as a single squashfs file and unpacked. The benefits are that you get symlinks preservation. This is because to avoid packaging the full `php` binary in each Photon application, I only symlink it to the required version:

```
$ ls /opt/phpfarm/inst/
bin  php-5.3.5  php-5.3.8  php-5.3.9
```

This of course means that I need on each node to have the right PHP version installed for the given process. This is not really a problem as a new runtime application node can be provisioned, installed and configured in 20 minutes automatically from empty drives to running in the cluster.

The applications are running und the user `nobody`, as such, they cannot write on disk. This means that for large binary files or uploads a system like Amazon S3 must be used. For the same reasons, sessions must be stored in the database. This *share nothing* architecture is the key to enable easy scaling.

At the moment, old slugs are not removed, but for each application, the current slug is tracked, with its UUID and creation time. It would be easy to parse these files to perform a bit of cleaning.

```
$ cat /var/run/bareku/apps/chemeo.def
2765e91b-50c4-4d04-b92c-f38f82661b2a 1327466830
```

For the rollbacks, this is really simple, just commit a new version and push it — git revert is here to help you.

For the *database migrations*, you have two solutions:

1. bring the apps down, upgrade the DB;
2. write your application in a way to be able to manage different versions of the database.

Since working on Bareku, the second approach is my approach, this also means that I work without ORM for PostgreSQL and enjoy the flexibility of MongoDB.

diprocd, the Chef d'Orchestre

How all the processes are controlled? We developed the wonderful diprocd — DIstributed PROcess Control Daemon — a set of tools (worker, client and master) to distribute the list of processes to run (client, master) and run/monitor them (worker).

So, the only important part left in the build is the diprocd configuration:

```
$ ls /var/lib/bareku/
diprocd-client.json
diprocd-worker.json
```

The distributed process manager diprocd is very robust and decoupled. On each runtime node, a worker takes care of supervising the processes and a client updates the worker configuration based on notifications from the master (ZeroMQ PUB/SUB).

```
$ cat /var/lib/bareku/diprocd-client.json
{"pid_file": "/var/run/diprocd-client.pid",
 "log_file": "/var/log/diprocd-client.log",
 "master_stats": "tcp://192.168.1.110:31123",
 "master_updates": "tcp://192.168.1.110:31124",
 "node_name": "%H",
 "conf_file": "/var/lib/bareku/diprocd-worker.json"}
```

The `%H` means that the `node_name` is effectively the hostname. This is nice because it means we have the exact same configuration on all the runtime nodes.

The client is simply subscribing to the master publication addressed to its hostname. It gets from their the new list of processes to manage and updates the worker configuration accordingly. The worker detects the change in configuration and stop, start or reload the processes to run the new version. The worker is smart, it can die and restart without affecting the running processes.

Frontend Node, Mongrel2

The frontend is really simple, it is basically *just* Mongrel2 listening to the requests and pushing them to the handlers. The handlers are publishing their answers back to Mongrel2 for it to forward them to the clients.

This is basically a zero maintenance system. Mongrel2 is robust and can easily push 1000's of requests per seconds. With the scale of our applications, we are on the safe side.

We also have a path to out grow a single frontend node. You simply do like Google, you get your DNS server to return several IPs for your domain, each one served by a distinct frontend node. Each frontend node can then be connected to the same *cluster* of handlers or distinct ones. This way you can easily provide multizone availability. You will have hard time reaching the limits of your hardware. Basically, your database will stop you before.

Applied Principles

- It is **fun** and **painless**, you just push and see the new version online.
- It has **minimal coupling**, if diprocd dies, everything keeps going, if the repository hosting node dies, it keeps going, if an application node dies most of the stuff will keep working, if the front end dies everything is offline. Oups. :D
- It runs **trusted applications** very well.
- It is **scalable** as each component can be scaled out to keep up with the load at **nearly constant latency**.

- It supports **zero downtime update** for the application, as the requests will just be buffered a bit longer the time to get the new handlers up and running.
- It is **relatively secure** as everything is done with encryption and without passwords. The applications are running with the minimal number of rights, usually under the `nobody` user.

External Services

The Ganeti cluster is also providing a range of services:

- PostgreSQL with a warm standby;
- MongoDB in replicaset;
- Mailserver with DKIM;
- Memcached;
- CDN/S3 *clone*;
- DNS server;
- File servers.

These services are for some of them taking benefits of the PaaS, for example the S3 *clone* is running on top of the PaaS and MongoDB. We are not storing a large amount of data in it, this is mostly used as a convenient CDN.

Conclusion

It is definitely a good idea to setup your own PaaS. The constraints will force you to code your applications a bit better and the ease of deployment will make you want to code more. Also, even if your PaaS is built to handle *generic* applications, you can do way more, but the decoupling of the three components: *front end*, *applications* and *services* is making scaling easier and simpler.

You must notice that **Mongrel2** and **ZeroMQ** are particularily well adapted as they naturally fit a service oriented architecture. Try them now, just to know about them for your next project. They may save your day.

Copyrights

All the notes are copyright © 2011 Céondo Ltd, all rights reserved. The latest version is available on Céondo's Technology Notes server.