Build reliable, traceable, distributed systems with ZeroMQ

PYCON2012

@jpetazzo
@dot_cloud

dotCloud

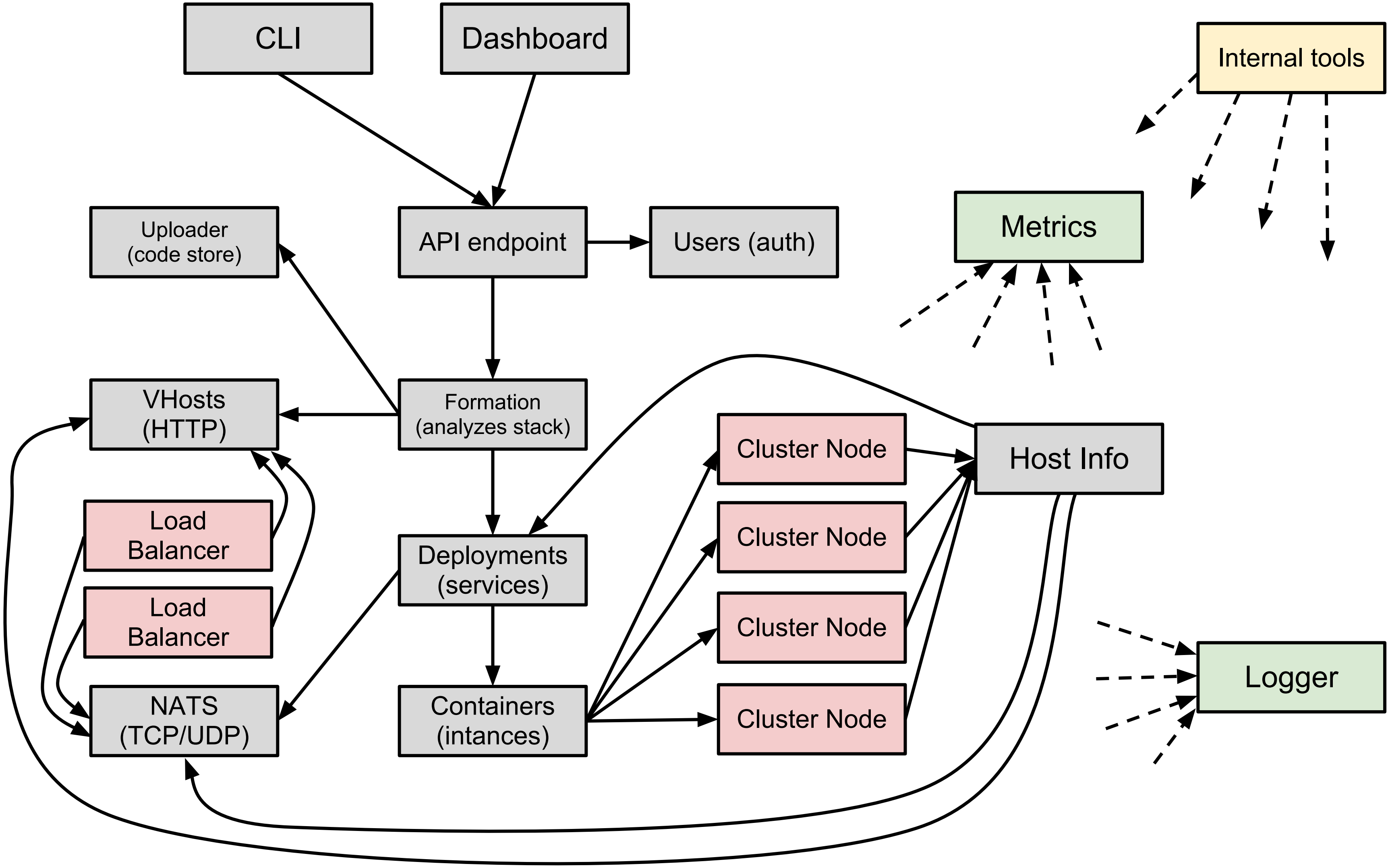
Outline

- Who we are
- Quick tour of ZeroRPC features
- Implementation overview

Introduction: why?

- We are running a PAAS:
  we deploy, monitor, and scale your apps
  (in the Cloud!)
- Many moving parts
- ... On a large distributed cluster

dotCloud architecture (simplified) — components, servers, things receiving data from everything else, me



Our requirements (easy)

- We want to be able to expose arbitrary code, with minimal modification (if any)

If we can do **import foo; foo.bar(42)**,
we want to be able to do **foo=RemoteService(...) ; foo.bar(42)**

- We want a self-documented system

We want to see methods, signatures, docstrings, without opening the code of the remote service, or relying on (always out-dated) documentation

Our requirements (harder)

- We want to propagate exceptions properly
- We want to be language-agnostic

- We want to be brokerless, highly available, fast, and support fan-in/fan-out topologies

(Not necessarily everything at the same time!)

- We want to trace & profile nested calls

Which call initiated the subcall which raised **SomeException**?
Which subcall of this complex call causes it to take forever?

Why not {x} ?

- x=HTTP
    - too much overhead for some use-cases (logs and metrics)
    - continuous stream of updates requires extra work (chunked encoding, websockets...)
    - asynchronous notification requires extra work (long polling, websockets...)
    - fan-out topology requires extra work (implementing some kind of message queue)
- x=AMQP (or similar)
    - too much overhead for some use-cases
    - requires a broker from the ground up

What we came up with

ZeroRPC!

- Based on ZeroMQ and MessagePack
- Supports everything we needed!
- Multiple implementations
  - Internal "reference" implementation in Python
  - Public "alternative" implementation with gevents
  - Internal Node.js implementation (so-so)

Example: unmodified code

Expose the "urllib" module over RPC:

```
$ zerorpc-client --server --bind tcp://0:1234 urllib
```

Yup. That's it.

There is now something listening on TCP port 1234, and exposing the Python "urllib" module.

Example: calling code

From the command line (for testing):

```
$ zerorpc-client  tcp://0:1234  quote "hello pycon"
"hello%20pycon"
```

From Python code:

```
>>> import zerorpc
>>> remote_urllib = zerorpc.Client( )
>>> remote_urllib.connect('tcp://0:1234')
>>> remote_urllib.quote('hello pycon')
'hello%20pycon'
```

Example: introspection

We can list methods:

```
$ zerorpc-client  tcp://localhost:1234 | grep  ^q
quote                          quote('abc def') -> 'abc%20def'
quote_plus                     Quote the query fragment of a URL; replacing ' ' with '+'
```

We can see signatures and docstrings:

```
$ zerorpc-client  tcp://localhost:1234  quote_plus  - ?

quote_plus(s, safe='')

Quote the query fragment of a URL; replacing ' ' with '+'
```

Example: exceptions

```
$ zerorpc-client  tcp://localhost:1234  quote_plus
Traceback (most recent call last):
  File "/home/jpetazzo/.virtualenvs/dotcloud_develop/bin/zerorpc-client", line 131, in <module>
    main()
  File "/home/jpetazzo/.virtualenvs/dotcloud_develop/bin/zerorpc-client", line 127, in main
    pprint(client(args.command, *parameters))
  File "/home/jpetazzo/Work/DOTCLOUD/dotcloud/zerorpc/zerorpc.py", line 362, in __call__
    return self.recv()
  File "/home/jpetazzo/Work/DOTCLOUD/dotcloud/zerorpc/zerorpc.py", line 243, in recv
    ret = self.handler(headers, method, *args)
  File "/home/jpetazzo/Work/DOTCLOUD/dotcloud/zerorpc/zerorpc.py", line 323, in handle_result
    raise e
TypeError: quote( ) takes at least 1 argument (0 given)
```

Example: load balancing

Start a load balancing hub:

```
$ cat  foo.yml
in:  "tcp://*:1111"
out: "tcp://*:2222"
type: queue
$ zerohub.py  foo.yml
```

Start (at least) one worker:

```
$ zerorpc-client  --server  tcp://localhost:2222  urllib
```

Now connect to the "in" side of the hub:

```
$ zerorpc-client tcp://localhost:1111
```

Example: high availability

Start a local HAProxy in TCP mode, dispatching requests to 2 or more remote services or hubs:

```
$ cat haproxy.cfg
listen zerorpc 0.0.0.0:1111
        mode tcp
        server backend_a localhost:2222 check
        server backend_b localhost:3333 check
$ haproxy  -f  haproxy.cfg
```

Start (at least) one backend:

```
$ zerorpc-client  --server  --bind  tcp://0:2222  urllib
```

Now connect to HAProxy:

```
$ zerorpc-client tcp://localhost:1111
```

Non-Example: PUB/SUB (not in public repo—*yet*)

- Broadcast a message to a group of nodes
  - But if a node leaves and rejoins, he'll lose messages
- Send a continuous stream of information
  - But if a speaker or listener leaves and rejoins...

You generally don't want to do this!

Better pattern: ZeroRPC streaming with gevents

Example: streaming

- Server code returns an ~~list~~ iterator
- Client code gets an ~~list~~ iterator
- Small messages, high latency? No problem!
  - Server code will pre-push elements
  - Client code will notify server if pipeline runs low
- Huge messages? No problem!
  - Big data sets can be nicely chunked
  - They don't have to fit entirely in memory
  - Don't worry about timeouts anymore
- Also supports long polling

Example: tracing (not in public repo—*yet*)

```
$ dotcloud  --trace  alias  add  sushiblog.web  www.deliciousrawfish.com
TraceID: 48aca4f4-75d5-40f2-b5bd-73c40ca40980
Ok. Now please add the following DNS record:
www.deliciousrawfish.com. IN CNAME gateway.dotcloud.com.

$ dotcloud-tracedump  48aca4f4-75d5-40f2-b5bd-73c40ca40980
[2012-03-07 23:56:17.759738] uwsgi@api --- run_command ---> api_00@zeroworkers (lag: 1ms | exec: 98ms)
[2012-03-07 23:56:17.770432] api_00@zeroworkers --- track ---> mixpanel_00@zeroworkers (lag: 1ms | exec: 70ms)
[2012-03-07 23:56:17.771264] api_00@zeroworkers --- km_track ---> mixpanel_01@zeroworkers (lag: 1ms | exec: 71ms)
[2012-03-07 23:56:17.771994] api_00@zeroworkers --- km_track ---> mixpanel_02@zeroworkers (lag: 1ms | exec: 71ms)
[2012-03-07 23:56:17.773041] api_00@zeroworkers --- record_event ---> users-events@users (lag: 0ms | exec: 0ms)
[2012-03-07 23:56:17.774972] api_00@zeroworkers --- info (pending) ---> formation-in [Not Received]
[2012-03-07 23:56:17.783590] api_00@zeroworkers --- info (pending) ---> formation-in [Not Received]
[2012-03-07 23:56:17.830107] api_00@zeroworkers --- add_alias ---> deployments_04@deployments (lag: 0ms | exec: 27ms)
[2012-03-07 23:56:17.831453] deployments_04@deployments --- exists_frontend ---> vhosts@vhosts (lag: 0ms | exec: 4ms)
[2012-03-07 23:56:17.836288] vhosts@vhosts --- OK ---> deployments_04@deployments (lag: 0ms | exec: 0ms)
[2012-03-07 23:56:17.837370] deployments_04@deployments --- exists ---> vhosts@vhosts (lag: 0ms | exec: 1ms)
[2012-03-07 23:56:17.838733] vhosts@vhosts --- OK ---> deployments_04@deployments (lag: 0ms | exec: 0ms)
[2012-03-07 23:56:17.840068] deployments_04@deployments --- add_frontend ---> vhosts@vhosts (lag: 0ms | exec: 15ms)
[2012-03-07 23:56:17.856166] vhosts@vhosts --- OK ---> deployments_04@deployments (lag: 0ms | exec: 0ms)
[2012-03-07 23:56:17.857647] deployments_04@deployments --- OK ---> api_00@zeroworkers (lag: 0ms | exec: 0ms)
[2012-03-07 23:56:17.859824] api_00@zeroworkers --- OK ---> uwsgi@api (lag: 0ms | exec: 0ms)
```

Implementation details

This will be useful if...

- You think you might want to use ZeroRPC
- You think you might want to hack ZeroRPC
- You want to reimplement something similar
- You just happen to love distributed systems

ØMQ

- Sockets on steroids
  http://zguide.zeromq.org/page:all
- Handles (re)connections for us
- Works over regular TCP
- Also has superfast ipc:// and inproc://
- Different patterns:
  - REQ/REP (basic, synchronous RPC call + response)
  - PUB/SUB (shout and listen to streams of events)
  - PUSH/PULL (load balance or collect messages)
  - DEALER/ROUTER (REQ/REP with routing)
- **pip install pyzmq-static** FTW (thanks @brandon_rhodes!)

Serialization: MessagePack

In our tests, msgpack is more efficient than JSON, BSON, YAML:

- 20-50x faster
- serialized output is 2x smaller or better

```
$ pip install msgpack-python
```

```
>>>  import msgpack
>>> bytes = msgpack.dumps(data)
```

Wire format

Request: (headers, method_name, args)

- headers dict
  - no mandatory header
  - carries the protocol version number
  - used for tracing in our in-house version
- args
  - list of arguments
  - no named parameters

Response: (headers, ERR|OK|STREAM, value)

Timeouts

- ØMQ does not detect disconnections
(or rather, it works hard to hide them)
- You can't know when the remote is gone
- Original implementation: 30s timeout
- Published implementation: heartbeat

Introspection

- Expose a few special calls:
  - _zerorpc_list to list calls
  - _zerorpc_name to know who you're talking to
  - _zerorpc_ping (redundant with the previous one)
  - _zerorpc_help to retrieve the docstring of a call
  - _zerorpc_args to retrieve the argspec of a call
  - _zerorpc_inspect to retrieve everything at once

- Introspection + service discovery = WIN

Naming

- Published implementation does not include any kind of naming/discovery
- In-house version uses a flat YAML file, mapping service names to ØMQ addresses and socket types
- In progress: use DNS records
  - SRV for host+port
  - TXT for ØMQ socket type (not sure about this!)
- In progress: registration of services
  - Majordomo protocol

Security: there is none

- No security at all in ØMQ
  - assumes that you are on a private, internal network
- If you need to run "in the wild", use SSL:
  - bind ØMQ socket on localhost
  - run stunnel (with client cert verification)
- In progress: authentication layer
- dotCloud API is actually ZeroRPC, exposed through a HTTP/ZeroRPC gateway
- In progress: standardization of this gateway

Tracing (not published *yet*)

- Initial implementation during a hack day
  - bonus: displays live latency and request rates, using http://projects.nuttnet.net/hummingbird/
  - bonus: displays graphical call flow, using http://raphaeljs.com/
  - bonus: send exceptions to airbrake/sentry
- Refactoring in progress, to "untie" it from the dotCloud infrastructure and Open Source it

How it works: all calls and responses are logged to a central place, along with a ***trace_id*** *unique to each sequence of calls.*

Tracing: trace_id

- Each call has a **trace_id**
- The **trace_id** is propagated to subcalls
- The **trace_id** is bound to a local context (think thread local storage)
- When making a call:
  - if there is a local **trace_id**, use it
  - if there is none ("root call"), generate one (GUID)
- **trace_id** is passed in all calls and responses

Note: this is not (yet) in the github repository

Tracing: trace collection

- If a message (sent or received) has a **trace_id**, we send out the following things:
  - **trace_id**
  - call name (or, for return values, OK|ERR+exception)
  - current process name and hostname
  - timestamp

*Internal details: the collection is built on top of the standard **logging** module.*

Tracing: trace storage

- Traces are sent to a Redis key/value store
  - each **trace_id** is associated with a list of traces
  - we keep some per-service counters
  - Redis persistence is disabled
  - entries are given a TTL so they expire automatically
  - entries were initially JSON (for easy debugging)
  - ... then "compressed" with msgpack to save space
  - *approximately* 16 GB of traces per day

Internal details: the logging handler does not talk directly to Redis; it sends traces to a collector (which itself talks to Redis).

The problem with being synchronous

- Original implementation was synchronous
- Long-running calls blocked the server
- Workaround: multiple workers and a hub
- Wastes resources
- Does not work well for *very long* calls
    - Deployment and provisioning of new cluster nodes
    - Deployment and scaling of user apps

Note: this is not specific to ZeroRPC
(Preforking servers, threaded servers, WSGI...)

First shot at asynchronicity

- Send asynchronous events & setup callbacks
- "Please do **foo(42)** and send the result to this other place once you're done"
- We tried this. We failed.
  - distributed spaghetti code
  - trees falling in the forest with no one to hear them
- Might have worked better if we had...
  - better support in the library
  - better naming system
  - something to make sure that we don't lose calls (a kind of distributed FSM, maybe?)

Gevent to the rescue!

- Write synchronous code
  (a.k.a. don't rewrite your services)
- Uses coroutines to achieve concurrency
- No fork, no threads, ~~no problems~~
- Monkey patch of the standard library
  (to replace blocking calls with async versions)
- Achieve "unlimited" concurrency server-side

The version published on github uses gevent.

Show me the code!

https://github.com/dotcloud/zerorpc-python

```
$ pip install git+git://github.com/dotcloud/zerorpc-python.git
```

Has: **zerorpc** module, **zerorpc-client** helper, exception propagation, **gevent** integration

Doesn't have: tracing, naming, helpers for PUB/SUB & PUSH/PULL, authentication

Questions?

Thanks!