CCNQ(4)

Stéphane Alnet

RMLL, July 2015

CCNQ was born in 2006.

That's 9 years ago.

What happened during these 9 years?

Let's go Historical

v0.x (2006) ssh push

(think ansible)

- System and customer configuration in a single document.
- Only OK when you're doing a couple changes a day.
- Code was lost in the Carribeans.

v1.5 (2007) Perl (sync)

CouchDB?

- Barely better.
- Code also thankfully lost.

v2.0 (2009) Perl async, CouchDB

- CouchDB on each call-processing server
- Perl async for realtime change management
- Task management via CouchDB
- Everything: provisioning, rating, UI, ...
- Perl packages

Doing async in Perl is hard.

- No unified model for callbacks.
- Incompatibilities between modules sometimes create hard-to-resolve locking situations.
- Never was able to get a CouchDB/AMQP/FileSystem synchronization to work properly.

v3 (2010) Node.js, CouchDB, RabbitMQ

- Focus on deployable call-processing
- Node.js for async
- Debian packages
- AMQP for M2M
- Host configurations in CouchDB
- CouchDB-based DNS

Host configuration changes are dynamically applied (e.g. FreeSwitch XML configuration files generated on the fly + `reloadxml`).

Debian packages are hard.

AMQP is verbose. (Also: 1ko limit per message?)

v4 (2014) Node.js, CouchDB, Docker.io, Socket.IO

- Docker.io: one app, one container.
- Apps built out of tiny components.
- package.json & git for (strong) dependency management.
- Socket.io for M2M & UI

- Host configurations in git(lab)

Host configurations in git(lab) really is a workaround. Ideally should move back to a database-centric approach to host provisioning. Lots of code to migrate from CCNQ3, still iterating.

Let's talk About Code

Guiding Principles

- APIs: REST/JSON and Socket.IO only
  No HTML generated on servers, use templating on the client.
- Prefer `changes` over directives
- Same modules on the server and the client: PouchDB, Socket.io-client, superagent, bluebird (Promises)

Promises

This isn't 2011 anymore.

```
serialize cfg, 'config'
```

Configure using `config` middlewares.

```
.then ->
  unless cfg.server_only is true
    fs.writeFileAsync process.env.FSCONF, xml, 'utf-8'
```

Write FreeSwitch XML configuration (if needed)

```
.then ->
  supervisor.startProcessAsync 'server'
```

Start the call-handler service

```
.then ->
  unless cfg.server_only is true
    supervisor.startProcessAsync 'freeswitch'
```

Start FreeSwitch (if needed)

```
.then ->
  debug 'Done'
```

Source: thinkable-ducks/config

Fluent

```
SuperAgent
.get "#{@cfg.auth_base ? @cfg.proxy_base}/_session"
.accept 'json'
.auth user.name, user.pass

.then ({body}) =>
  @session.couchdb_username = body.userCtx.name
  @session.couchdb_roles = body.userCtx.roles
  @session.couchdb_token = hex_hmac_sha1 @cfg.couchdb_secret, @session.couchdb_username
```

Source: spicy-action/couchdb-auth Uses package `superagent-as-promised`
Also illustrates combining fluent interfaces with Promises.

Domain-Specifc LanguagesDSL

```
require('zappajs') cfg.web, ->

  @get '/', ->
    @json
      ok: true
      name: pkg.name
      version: pkg.version

  db = new PouchDB cfg.db

  @get '/forwarding/:number', ->
    db.get @param.number
    .then (doc) =>
      @json forwarding: doc.forwarding

cfg = require process.env.CONFIG ? './local/config.json'
pkg = require './package.json'
PouchDB = require 'pouchdb'
db = new PouchDB cfg.db
```

```
require('zappajs') cfg.web, ->

  @on trace: ->
    if @session.admin
      @broadcast_to 'trace-servers', 'trace', @data
```

```
client = require('socket.io-client') process.env.SOCKET

client.on 'trace', (doc) ->
  client.emit 'trace_started', host:hostname, in_reply_to:doc
  Promise.resolve()
  .then ->
    trace doc
  .then ->
    client.emit 'trace_completed', host:hostname, in_reply_to:doc
  .catch (error) ->
    client.emit 'trace_error',
      host: hostname
      in_reply_to: doc
      error: error
```

Source: project `nifty-ground`. Note the trick with `Promise.resolve()` which allows to start the Promise chain whether function `trace` returns a Promise or not.

Domain-Specifc LanguagesVoicemail

```
class User

  main_menu: ->
    @call.get_choice "phrase:voicemail_main_menu"
    .then (choice) =>
      switch choice

        when "1"
          @retrieve_new_messages()
          .then (rows) =>
            @navigate_messages rows, 0
          .then =>
            @main_menu()

        when "3"
          @config_menu()
```

Souce: project `well-groomed-feast`, with some renaming to simplify.

Let’s talk aboutFun

`esl` module

- provides *client* access to FreeSwitch events socket
  e.g. to build a dialer
- provides *server* access to FreeSwitch events socket
  inbound call handling

Success Story

- used in production and to build new services

Testability

- `esl` has both unitary tests and live tests
- live tests involve starting and stopping FreeSwitch: much easier with Docker.io!

Middleware `useful-wind`

- Take the middleware concept from Connect / Express / Zappa
- Apply it to voice calls
- Build call-processing applications by combining (npm) modules

Middleware Power `thinkable-ducks`

- process calls (ESL)
- access CouchDB (PouchDB)
- receive and send events (Socket.IO-client)
- serve APIs (ZappaJS)

Examples:

- `tough-rate` LCR engine used in production at K-net
- `well-groomed-feast` voicemail engine

Distributed Sniffer `nifty-ground`

Browser

- Request generated by JS on the browser
- Sent over Socket.io to dispatcher.

Server

- Process on each server waits for request from dispatcher
- Send notification → browser builds list of expected responses
- Queries captures files
- Send notification (with data)
- Store PCAP file (last 500 packets) in CouchDB

More Fun

PouchDB

Browser:

```
db = new PouchDB 'users'

db.put _id:'shimaore', name:'Stéphane Alnet'
.then ->
  db.get 'shimaore'
.then (doc) ->
  assert doc.name is 'Stéphane Alnet'
.catch (error) ->
  cuddly.csr "Could not retrieve shimaore: #{error}"
```

All accesses are local to the browser. Database is persisted.

Browser: Access CouchDB

```
db = new PouchDB 'https://couchdb.example.net:6984/users'
```

Sync Browser ←→ CouchDB

```
PouchDB.sync 'users', 'https://couchdb.example.net:6984/users'
```

Two-way replication. Also exist as one-way replication with `replicate`. Offline-first made easy. Also check out Hoodie.hq!

Server: Use local database

```
db = new PouchDB 'users'
```

Docker.io

- In production we use `--network=host`
  Although it would help greatly document things if we were using the Docker connection thingies. But we do a lot of UDP and kernel-level stuff for speed.
- Lessons learned: need to consider containers as read-only images
  Otherwise they grow larger and larger in production due to AUFS. Use mountpoint for logs, live data.
- It's hard to use independent UIDs inside containers.
  When mounting logs etc the UIDs are kept identical, often resulting in access issues.
- `docker-ccnq` module (private) to ensure proper start of containers.
  Essentially git pull at install + `for dir in ~docker/start/[0-9]*; do cd $dir && ./init start; done`
- Large amount of disk vs compression.
  Docker.io uses large amounts of disk because the intermediary (build) steps of a Dockerfile are part of the final image. Compression (=keep only data that is still present) is a hotly debated topic.
- Avoid using `latest` tags.
  Same as when dealing with dependencies without Semantic Versioning: you don't know what you are actually deploying when you deploy `latest`.

Here's a typical deployment `init` script like the ones we currently use in production:

```
#!/bin/bash
CONFIG=/opt/thinkable-ducks/config.json
REGISTRY=(redacted)
VERSION=3.6.5
REPO=shimaore/tough-rate:${VERSION}

case "$1" in
  pull)
    docker pull "${REGISTRY}/${REPO}"
    ;;

  start)
    { echo -n "#### Start $DOCKER_NAME $REPO ####"; date; git show; } >> $HOME/version.log
    # Remove any lingering container.
    docker rm ${DOCKER_NAME} || echo '(ignored)'
    # Create log directory if it doesn't exist.
    mkdir -p log
    # Start the image.
    docker run -d --net host \
      --restart=always \
      --name ${DOCKER_NAME} \
      --env-file=./env \
      -v ${PWD}/config.json:${CONFIG} -e CONFIG=${CONFIG} \
      -v ${PWD}/log:/opt/tough-rate/log \
      "${REGISTRY}/${REPO}"
    ;;

  stop)
    { echo -n "#### Stop $DOCKER_NAME $REPO ####"; date; } >> $HOME/version.log
    docker kill ${DOCKER_NAME} || echo '(ignored)'
    docker rm   ${DOCKER_NAME} || echo '(ignored)'
    true
  ;;
```

JSON Swiss Army KnifeJQ

```
rest -G \
  -d startkey='\"rule:16171\"' \
  -d endkey='\"rule:1617999\"' \
  https://couchdb.example.net/ruleset/_all_docs | \

jq '{docs: (.rows | map({ _id: .id, _rev: .value.rev, _deleted:true })) }'  \

rest -X POST --data-binary @- \
  https://couchdb.example.net/ruleset/_bulk_docs
```

The command-line is alive and well!

`rest` is:

```
curl -n \
  -H 'Accept: application/json' \
  -H 'Content-Type: application/json' \
  "$@"
```

And more fun

- Project Metadata - `package.json`
  applies to npm, but I also use it in the Makefile that builds Docker images to figure out the version (`tag`) to apply to the docker build
- Semantic Versioning - http://semver.org/
- Dependency locking
  applies to `npm` and others, e.g. tshark in nifty-ground's Dockerfile
- Dependencies management: `npm`, `Dockerfile`; Gemnasium

Thank you | Merci

- **CCNQ4** https://github.com/shimaore/ccnq4
- **Code** https://github.com/shimaore/
- **Presentation**: http://shimaore.github.io/2015-rmll-dev
- **Contact** http://stephane.shimaore.net/