Ready to Go ?

[Retour d'expérience sur un projet en Golang]

HTML
http://goo.gl/nPljiO
PDF
http://goo.gl/gzgDZW

Qui sommes nous ?

Qui sommes nous ?

- Orange

Orange est le 3ème opérateur mobile et 1er fournisseur d'accès Internet ADSL en Europe, France Télécom compte parmi les leaders mondiaux des services de télécommunications aux entreprises multinationales.

- 100 000 salariés
- 7,4 millions de Livebox
- 9 millions de clients équipés haut débit, soit 46,3% de part de marché Grand Public
- 26,2 millions de clients mobiles dont 14,6 clients haut débit mobile
- 7,2 millions de clients VOIP
- 1200 boutiques France Télécom
- 736 000 clients Orange TV
- 380 000 clients ont signé pour la Fibre (644 000 foyers connectables)

Qui sommes nous ?

- Orange

- OAB

Orange Applications for Business

Machine to Machine, Internet des objets, big data… Orange Business Services rassemble dans un même pôle ses expertises : Orange Applications for Business.

- 2400 collaborateurs (+200 recrutements par an)
- Chiffre d’affaires de 300 millions d’euros en 2013 (sur un CA global de 6,5 milliards d’euros réalisé l’an passé par OBS)

Qui sommes nous ?

- Orange
- OAB
- L'équipe

L'équipe des développeurs ayant participé au projet est constituée de :

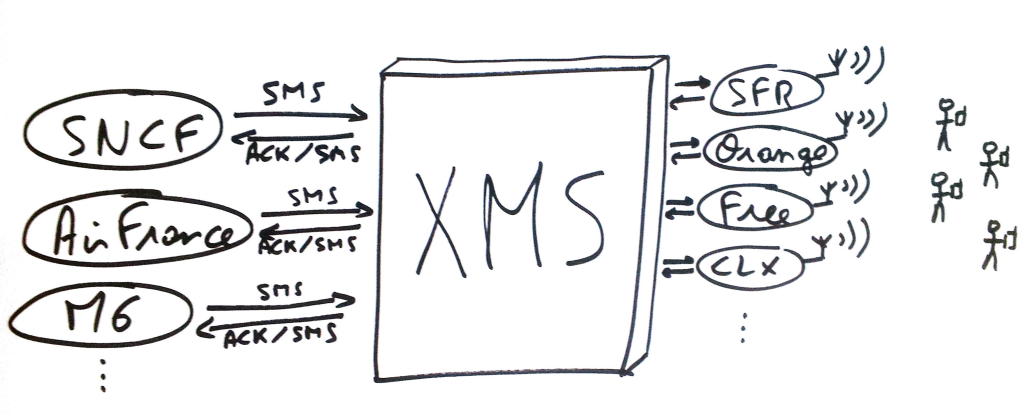
- Michel Casabianca
  michel.casabianca@orange.com
- Benjamin Chenebault
  benjamin.chenebault@orange.com
- Jacques Antoine Massé
  jacquesantoine.masse@orange.com
- Sébastien Font
  sebastien.font@orange.com

Qui sommes nous ?

- Orange
- OAB
- L'équipe
- La plate-forme XMS

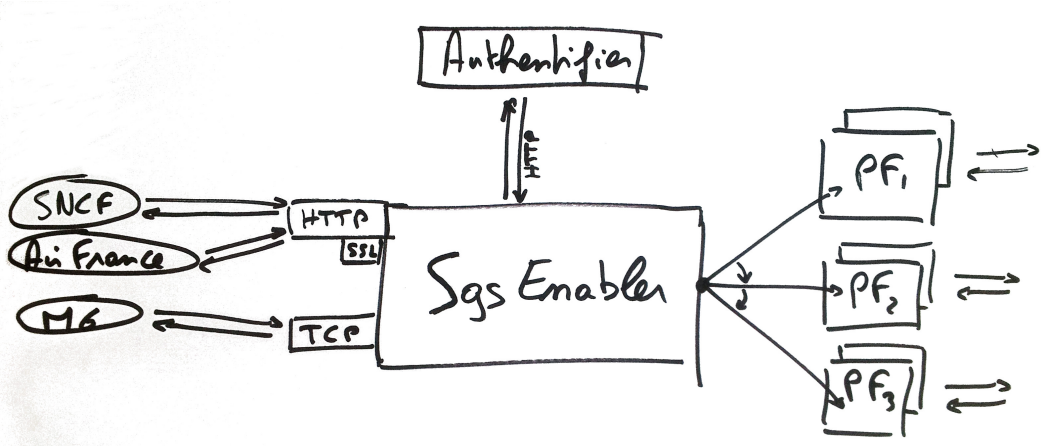
Plate-forme d'envoi et réception SMS/MMS

Entre des éditeurs de services et des usagers mobile



- ~ 30 applis en production
- Langages Java, C & Python
- 6 dev, 3 ops
- Plusieurs centaines de clients
- 900 millions de sms/an

Le frontal d'accès à la plateforme XMS



- Serveur TCP & HTTP
- Module d'authentification
- Routeur
- Load balancer

Maintenance très complexe et coûteuse

- Développé par un grand nombre de personnes
- Agrégat de design patterns : Observer, Factory, Object pool, Composite, [...]
- Très peu, voire aucune documentation
- Beaucoup de problématiques réseaux
- Problématiques d'accès concurrents réglés avec les classes de `java.util.concurrent`, et des blocs synchronisés
- Monitoré à partir de beans exposés en JMX

Conclusion

Malgré des mois passés à débugger l'application, elle n'a jamais été suffisamment stable pour pouvoir y migrer tous nos clients

Il a donc été envisagé de réécrire l'application

Etude technique

[Le choix de la technologie]

Le périmètre de l'étude technique

- Un seul connecteur (frontal HTTP)
- Fonctionnalités principales
  - Parsing XML
  - Authentification par IP
  - Appel d'un serveur par TCP

Critères de choix de la technologie

- Simplicité de développement
- Maintenance facile du code
- Performances au runtime
- Consommation ressources CPU/mémoire

Alternatives techniques

L'existant a été développé avec l'API Non-blocking I/O du JDK.

Les cibles envisagées ont été les suivantes :

- Java avec utilisation d'IO synchrones
- Go avec utilisation des channels et de goroutines

Les deux POCs ont été développés en parallèle en 10 jours environ

Les résultats des POCS

- Nombre de lignes de code comparable
- Architecture moins complexe en Go
- Tests en charge en faveur de Go (10% environ)

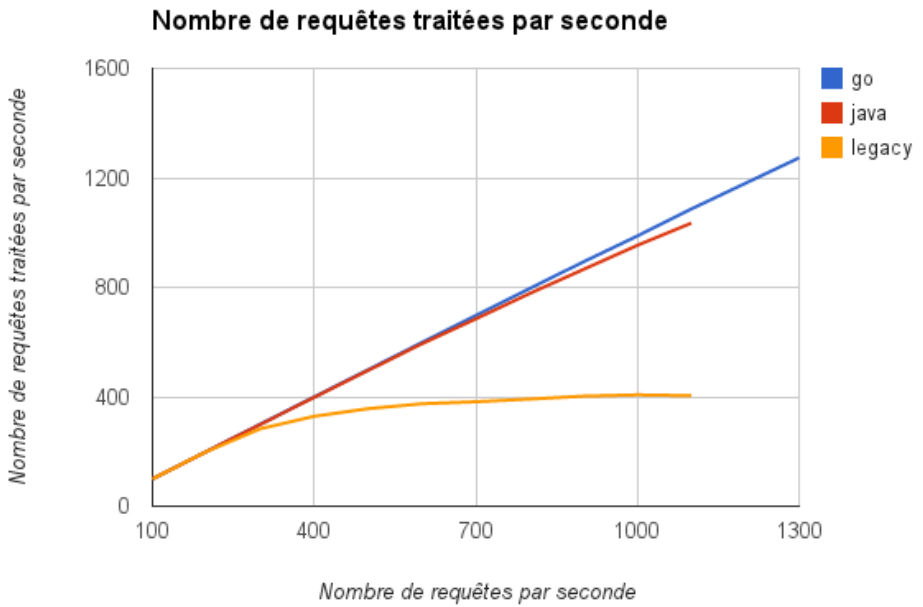
Performances mesurées

- Nb de requêtes par seconde
- Temps moyen de traitement d'une requête

Etude technique

- Périmètre
- Critères de choix
- Les alternatives
- Résultats
- Nombre de requêtes par seconde

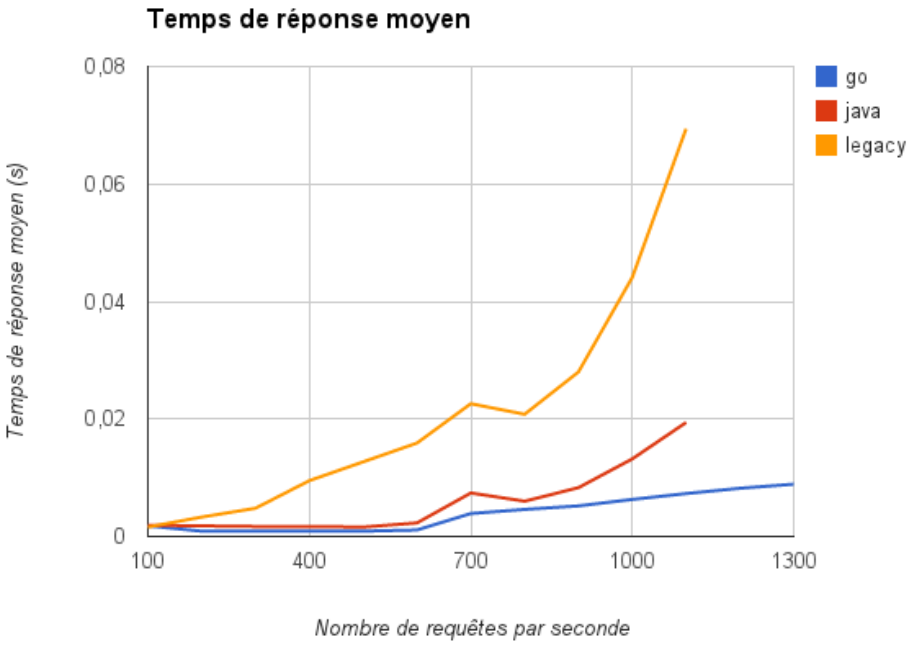
Nombre de requêtes par seconde



Etude technique

- Périmètre
- Critères de choix
- Les alternatives
- Résultats
- Nombre de requêtes par seconde
- Temps moyen par requête

Temps moyen par requête



Le langage Go

Go est un langage :

- Créé par Google en 2007, v1.0 en mars 2012
  - R.Pike, K.Thompson, R. Griesemer
- Procédural, un peu objet, un peu fonctionnel
- Garbage collecté
- Compilé
- Typage fort, statique
- Orienté concurrence
- Open source

Les channels

```
package main

func main() {
    c := make(chan int, 1)
    c <- 42
    val := <-c
    println(val)
}
```

http://play.golang.org/p/Kq0Ih_NwIH

- Primitives du langage
- File FIFO
- Très largement utilisée pour gérer la concurrence et les attentes de thread
- "select" permet de poller plusieurs channels

Les Goroutines

- Exécution d'un appel de fonction en concurrence
- Mot clé "go"
- Primitive du langage
- Faible occupation mémoire (~4ko/goroutine)
- Task switching peu significatif
- Multiplexé sur un ou plusieurs threads de l'OS

Exemple

```
package main

func producer(c chan int) {
    for i:=0; i<5; i++ {
        c <- i
    }
}

func consumer(c chan int) {
    for {
        i := <-c
        println(i)
    }
}

func main() {
    c := make(chan int)
    go consumer(c)
    producer(c)
}
```

http://play.golang.org/p/Bu2lvD5NCO

Les commandes Go

- En ligne de commande
- go build
  - Compilation
- go run
  - Compilation + exécution
- go test
- go get

[...]

Les exécutables

- Binaires sans dépendance dynamique
- Volumineux
  - "Hello world" ~ 1 Mo
  - Notre application ~ 9 Mo
  - Embarque toutes les bibliothèques utilisées
- Plate Formes supportées :
  - FreeBSD et Linux 32/64 sur x86 et ARM, Windows, MacOS,...

L'environnement de développement

- go code
- Existence de modes pour emacs et vi :
  - go-vim
  - go-snippets, autocomplete, flycheck, etc.
- Liteide
  - Open Source
  - Ecrit en Go
- Plugins Eclipse et IntelliJ peu matures.
- Navigation dans le source, renommage, code appelant, build/tests, etc.

Les écueils

La gestion des erreurs est rébarbative

Source Go typique :

```
f, err := os.Open("filename.ext")
if err != nil {
    log.Fatal(err)
}
```

Cette gestion des erreurs :

- Est répétitive
- On ne peut gérer des erreurs *en bloc*
- On ne peut typer les erreurs

Il est possible de lancer des *paniques* :

- Elles sont propagées
- Elles peuvent être interceptées
- Ce ne sont cependant pas des exceptions

APIs simplistes

L'API de logs est assez critiquée car elle :

- Ne gère pas des niveaux de logs
- Ne gère pas des fichiers de configuration
- Doit donc être configurée dans le code

L'API de parsing des options en ligne de commande ne respecte par les standards Unix.

Gestion des encodages

Seul l'*UTF-8* et l'*UTF-16* sont supportés.

Ce choix est évident, mais peut rendre difficile la gestion de l'existant.

Le versioning

- `go get` clone le dernier commit des repo GitHub, Bitbucket, Google code
- Absence volontaire de package manager natif
- Package managers développés par la communauté : gopack, godep, GoManager, dondur,

```
[deps.memcache]
import = "github.com/bradfitz/gomemcache/memcache"
tag = "1.2"

[deps.mux]
import = "github.com/gorilla/mux"
commit = "23d36c08ab90f4957ae8e7d781907c368f5454dd"
...
```

TLS immature

Nous avons rencontré des difficultés avec l'implémentation TLS :

- Des certificats générés sans l'option `ExtendedKeyUsage` (valeur `clientAuth`) ne peuvent servir à authentifier un client
- L'algorithme MD5 n'est pas supporté pour la signature de certificats bien qu'il soit dans la liste des algorithmes supportés par TLS 1.2
- Le handshake SSLv2 n'est pas supporté ce qui pose problème avec nombre de clients SSL (en particulier Java 1.6)

Si tous ces choix sont probablement pertinents du point de vue sécurité, ils peuvent poser des problèmes de compatibilité avec l'existant.

Nous avons résolu le problème en déléguant la gestion du TLS à un HA-proxy en façade.

Les Bonnes Surprises

Montée en compétence rapide

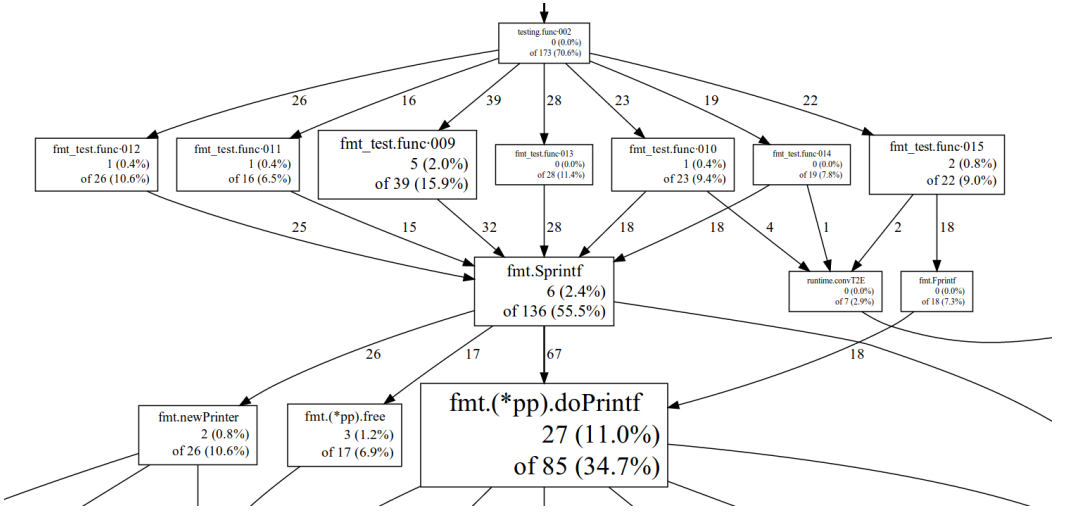
- La syntaxe est simple
  - "Langage procédural à accolades"
- Outillage efficace
- Goroutines et channels
  - Patterns de concurrence
- Features avancées
  - Composition de structures
  - Programmation "fonctionnelle"
  - Polymorphisme

Qualité des API

- Accessibles et compréhensibles
- API Standard complète
  - compression, crypto, bases de données, encodage, html, io, log, net, reflection, testing, time, unsafe
- Limite le recours aux API tierces
- Styles de développement hétérogènes

Monitoring

- Utilisation du package pprof
- Génération de heap dump et de cpu profiling
- Le package se bind sur un serveur HTTP
- `go tool pprof`, outil de visualisation des pprofs



L'API de tests

- Utilisation du package testing
- Par convention, les fichiers de tests sont nommés XXX_test.go
- `go test` permet d'exécuter les tests
- Il existe deux types de test :
  - Les tests unitaires : `TestXxx(*testing.T)`
  - Les benchmarks : `BenchmarkXxx(*testing.B)`

```
import "testing"

func TestFunctionTralala(t *testing.T) {
    if "tralala" != "tralala" {
        t.Error()
    }
}
```

Simplissime mais efficace

Accès concurrents

- Possibilité de lancer les tests unitaires avec l'option `-race`.

Détection des accès concurrents à la mémoire.

- Egalement possible d'appliquer cette option à la compilation pour détecter les accès concurrents au runtime.

Ceci peut être utile si la couverture de test est faible, mais attention aux performances !

Stabilité de l'application

Au cours de nos développements et de nos tests de charge, nous n'avons jamais vu l'application s'arrêter.

- Pas de SegFault ni de core dump
- Les pointeurs existent
- Mais on ne peut les manipuler (pas d'arithmétique de pointeurs)

Bonnes surprises

- Qualité des APIs
- Monitoring
- API de tests
- Accès concurrents
- Stabilité de l'application
- Support et communauté

Support et communauté

- Documentation du langage, des APIs
- Blogs et publications de la core team
- go-nuts, stackoverflow, reddit, go newsletter, irc
- Évènements et captation de conférences

Open source

Google a joué pleinement le jeu de l'Open Source :

- La licence du logiciel est très ouverte (BSD)
- Code source très clair et facilement modifiable
- Nombreuses bibliothèques tierces
- Développement dynamique

Retour sur les performances

Poste de Développement

- Affranchissement des limitations réseau
- Mocks plus performants qu'implémentations réelles

Résultats

- 254 req./s pour la version en GO
- 139 req./s pour la version en Java

Préproduction

Limitations dues :

- Au réseau
- Aux performances des applications connexes

Résultats

- 30 req./s pour la version en GO
- 30 req./s pour la version en Java (avec perte de paquets)

RAM et CPU

- Environnement de préproduction
- A charge égale de 30 req./s
- Java : 94% CPU, 8.5% RAM
- Go : 2% CPU, 1.2% RAM

Conclusion

Expérience concluante

Projet en production

Un langage syntaxiquement et conceptuellement simple

Adapté pour des applications pour lesquelles la performance est un enjeu

Outillage très simple à utiliser

Outils de profiling

Un vrai concurrent à Java…