# Avant-propos

Le travail de fin d'études (*TFE*) à l'EPHEC est un travail personnel réalisé tout au long du dernier quadrimestre du cursus académique. À travers ce travail, il nous est demandé de démontrer que nous avons acquis les connaissances techniques, les capacités d'analyse et de mise en oeuvre ainsi que la maturité requises à la réalisation d'un travail d'ampleur.

Le travail de fin d'études vaut 15 ECTS (*European Credits Transfer System*) et constitue à lui seul un quart des crédits de la troisème année, soit de 300h à 450h de travail.

En même temps que le travail de fin d'études, durant le dernier quadrimestre des études, nous sommes stagiaire à temps plein en entreprise. Mon travail de fin d'études s'est donc réalisé en soirée et durant mes weekends.

# Remerciements

Tout d'abord je tiens à remercier ma famille pour tout le soutien qu'elle m'a apporté tout au long de mon stage et de mon travail de fin d'études. Leur relecture attentive de mon rapport et leurs nombreux conseils m'ont beaucoup aidé à la rédaction de ce document.

Je tiens également à remercier l'ensemble des mes professeurs et particulièrement mon rapporteur M. Schalkwijk. Mon projet, Webgames, a évolué en même temps que moi entre les murs de l'EPHEC et nombreux sont les cours qui ont eu une influence sur les technologies employées et mes réflexions techniques.

Pour finir je remercie également mes collègues Devops à Stepstone qui ont pris le temps de répondre à mes questions concernant mon travail de fin d'études et qui ont pu me guider vers les ressources appropriées.

# Table des matières

# Introduction

L'utilisation toujours plus fréquente d'internet par de plus en plus de monde pose souvent problème dans les architectures traditionnelles. Les systèmes informatiques doivent être capables de traiter toujours plus d'information, et ce, toujours de manière plus fiable. Le développement de ces systèmes demande une grande rigueur de la part des développeurs et administrateurs systèmes qui doivent jongler entre stabilité du système et ajout de fonctionnalités.

De nombreuses techniques ont vu le jour ces dernières années autant pour le développement que pour le déploiement. En créant une véritable culture *DevOps*, les développeurs travaillent de plus en plus étroitement avec les administrateurs systèmes, chacun apportant sa culture et sa vision de l'informatique afin d'améliorer les produits finaux. En ce sens, la culture *DevOps* est une culture différente de ce qui se fait généralement en entreprise où les développeurs et administrateurs systèmes/réseaux sont dans des équipes différentes. La culture *DevOps* renforce la collaboration entre ces deux mondes : travail dans les mêmes locaux, même chef, mêmes réunions.

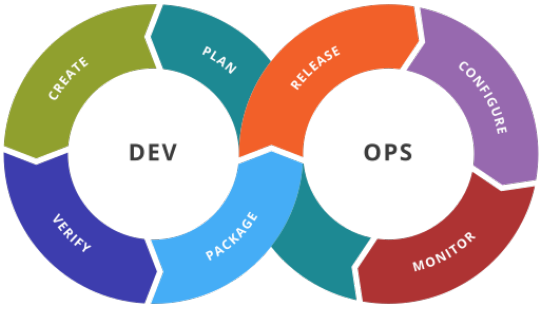

*Illustration 1: Cycle devops*

L'arrivé du *cloud* permet de nouvelles approches pour héberger des services, on parle par exemple d'architecture élastique où les machines sont allumées et éteintes en fonction de la demande. Si ce genre d'architecture devient de plus en plus prisée par les entreprises, le développement d'applications compatibles demande des approches différentes à la conception de systèmes traditionnels.

Cette problématique pose un problème de taille dans l'univers des jeux vidéo multi-joueurs en ligne, l'hébergement et l'évolutivité des serveurs présentent quelques challenges uniques. La plupart des jeux multijoueurs en ligne nécessitent un serveur de gestion dédié directement accessible sur Internet qui est à la fois le gardien du monde virtuel où les joueurs peuvent se connecter et jouer ensemble, ainsi que l'arbitre qui vérifie que personne ne triche.

C'est dans ce cadre que se place Webgames, mon projet, si des systèmes propriétaires existent chez les grands noms de l'industrie du jeu vidéo, Webgames veut combler le trou en proposant une solution d'hébergement élastique dans le cloud à destination des développeurs indépendants.

# Historique de mon projet - Webgames

Projet de recherche et terrain de jeu pour mes expérimentations, Webgames est un système qui a déjà connu plusieurs itérations de développement. Chaque itération (ou version) m'a permis de pratiquer le développement informatique, d'approfondir mes connaissances en réseau et de tenter différents modèles architecturaux parmi lesquels les systèmes distribués et les microservices.

## *Prémices - Jeu de cartes*

Les prémices de ma réflexion quant à l'hébergement de jeux ont commencé lorsque j'étais étudiant en première année à l'EPHEC. Je développais à l'époque un jeu de cartes disponible par navigateur. Une fois le jeu terminé, j'ai réfléchi à la manière de rendre ce jeu accessible aux joueurs. La solution mise en place est assez simple : le serveur accumule des joueurs dans une file d'attente et instancie une nouvelle partie dès que 4 joueurs sont connectés. Si cette solution pouvait gérer quelques parties en parallèle, il était cependant impossible de gérer une centaine voire un millier de parties en même temps. La puissance du serveur aurait dû être augmentée en ajoutant plus de mémoire et en améliorant le processeur; on parle ici d'évolutivité verticale (*vertical scalling*) : pour que le serveur soit capable de gérer plus, il lui faut du meilleur matériel physique. Ce système présente aussi l'important inconvénient que le moindre bug au niveau de n'importe quel jeu fait planter l'ensemble du système.

## *Webgames v1 - Multi-thread*

Durant les vacances d'été entre ma première et ma deuxième année à l'EPHEC, nous nous sommes attaqués au développement d'un jeu de Bomberman avec mon ami Mathieu Rousseau également étudiant en Technologie de l'Informatique, il s'occupait de toute la partie front-end (interface utilisateur et rendu du jeu) pendant que je me chargeais de la partie back-end (logique du jeu et hébergement). À l'inverse du jeu de cartes où les joueurs jouent chacun à leur tour, tous les joueurs jouent simultanément dans une partie de bomberman. Ayant appris du jeu de cartes, chaque partie était lancée dans un thread afin de l'isoler du reste du système : en cas de plantage du jeu, le système continue de fonctionner.

Malheureusement, l'utilisation d'un *GIL* (*Global Interpretor Lock*) dans CPython (l'implémentation la plus répandue de python) empêche les threads de tourner en parallèle. En python 2, on dénotait même des pertes de performances à l'utilisation des threads en python dues aux changements de contexte (*context switch*) inutiles, problème heureusement résolu depuis python 3.2.

## *Webgames v2 - Multi-process et Web Services*

Avec la v2 on introduisait le support pour plusieurs jeux, niveau back-end ce sont mes premiers pas dans le monde des systèmes distribués. En effet, le serveur monolith a été scindé en plusieurs processus, chacun responsable d'une tâche en particulier.

Un des processus était notre nouveau système d'authentification. Nous avons basé notre système sur *Yggdrasil*, le protocole d'authentification utilisé dans le jeu Minecraft. L'autorité d'authentification génère des jetons d'accès qu'elle communique aux utilisateurs, ceux-ci les utilisent pour accéder aux autres serveurs, serveurs qui à leur tour vérifient auprès de l'autorité que les jetons sont valides. Cette approche permet aux différents services d'authentifier l'utilisateur sans que les services n'aient besoin de connaitre son mot de passe.

L'autorité d'authentification se présentait comme une API Rest protégée par un système de bannissement temporaire automatique en cas d'erreurs de connexions successives.

Tout le code qui permettait de constituer des parties s'est également trouvé dans un processus à part qu'on a appelé *Match-Maker* pour suivre la philosophie UNIX : "*Do one thing and do it well*". Séparer différents composants d'une même application globale en plus petites unités permet une meilleure structure du code ce qui facilite à la fois la compréhension que la capacité à faire évoluer chaque composant indépendamment des autres.

Cette version de Webgames est également la seule qui a utilisé un *ORM* (*Object Relational Mapper*) au lieu d'exécuter directement des requêtes SQL. Un *ORM* est une couche logicielle qui se place entre l'application et la base de données, le développeur peut directement manipuler des objets sans avoir à se soucier des requêtes SQL. Les versions suivantes de Webgames seront baties sur un modèles asynchrone et *SQLAlchemy* (l'ORM) n'était compatible avec aucun driver asynchrone.

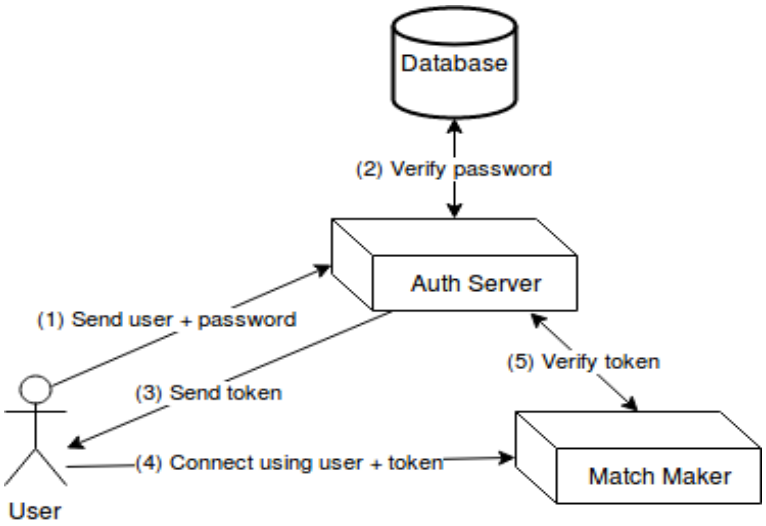


*Illustration 2: Yggdrasil*

## *Webgames v3 - Modèle réseau Asynchrone*

Cette version est à part, je n'ai fait que réimplémenter l'application de manière *asynchrone*. La programmation asynchrone est un modèle de développement qui a été popularisé par *JavaScript* (un autre langage de programmation), ce modèle se base sur les avancées des noyaux des systèmes d'exploitation récents qui permettent de gérer certaines opérations normalement *bloquantes* de manière *non-bloquantes*.

La lecture d'un fichier est un exemple d'opération bloquante, le processeur ordonne une lecture sur disque et attend que le fichier arrive en mémoire centrale pour continuer : pendant tout le temps de la lecture sur disque, le processus n'avance plus, il est bloqué. Avec les nouveaux systèmes, il est possible de démarrer la lecture sans avoir à attendre qu'elle se termine pour continuer, le processus peut effectuer d'autres actions en attendant que la lecture soit effectuée. Une fois celle-ci finie (et donc les données disponibles), le processus peut reprendre le traitement qui avait été mis en attente.

Ces appels systèmes permettent une certaine forme de programmation concurrente : lorsqu'une fonction est mise en attente, une autre fonction peut être exécutée jusqu'à ce qu'elle retourne une valeur ou qu'elle se mette également en attente d'une ressource. Pour que le processus puisse sauter d'une fonction asynchrone à l'autre, celles-ci sont enregistrées dans une *boucle événementielle* (*event loop*). Les fonctions sont alors exécutées, interrompues et reprises dans une symphonie guidée par la boucle.

Ce paradigme de programmation est à double tranchant, s'il permet de développer des systèmes très dynamiques et performants, il requiert que le système entier suive son modèle. En effet il suffit d'un seul appel système bloquant pour bloquer toute la boucle événementielle sur le traitement de cet appel et donc de bloquer tout le système. Un exemple est que je ne pouvais plus utiliser *SQLAlchemy*, un travail conséquent est encore aujourd'hui fourni par la communauté des développeurs python pour rendre compatibles les bibliothèques populaires mais le travail est conséquent.

De nombreux papiers ont été écrits quant à l'efficacité de l'approche asynchrone pour la gestion des connexions réseau. NGINX et HAProxy, deux logiciels réputés pour leur efficacité sont basés sur le modèle asynchrone, tous les tests comparatifs entre l'approche synchrone et asynchrone placent l'asynchrone en tête. Dédier plusieurs mois de recherche et d'efforts pour comprendre ce modèle et opérer à la réimplémentation de Webgames en a vraiment valu la peine.

## *Webgames v4 - Système Distribué partiellement Sans État*

### Stateless

Au fil de mes recherches, je suis tombé sur des systèmes à l'architecture *sans-état* (*stateless*). Être sans état signifie que l'application ne sauvegarde rien localement pour traiter des requêtes. Chaque requête est traitée indépendamment de toutes les requêtes précédentes et n'aura aucun impact sur les requêtes à venir. Être sans état signifie entre autres qu'aucune information ne subsiste localement entre chaque requête. *IP*, *UDP* et *HTTP* sont par exemple des protocoles sans état car aucun état n'est sauvé entre les requêtes. À l'inverse, *TCP* est un exemple de protocole avec état car une session régit la communication.

L'immense avantage qu'offrent les systèmes sans-états est leurs propriétés élastiques. Comme le serveur ne sauvegarde rien localement, il est possible d'en démarrer un second et de répartir la charge entre les deux serveurs. Il est même possible de monitorer la charge sur chacun des serveurs et d'ajouter ou retirer dynamiquement des serveurs à la *grappe* (*cluster*) afin d'avoir toujours autant de serveurs qu'il n'en faut pour gérer le trafic, on parle ici d'*élasticité horizontale automatique* (*horizontal auto-scalling*).

Grâce à *Yggdrasil*, le service d'authentification était déjà sans état. J'ai malgré tout préféré le ré-implémenter en utilisant les plus standards *JWT* (*JSON Web Tokens*). Yggdrasil est un système qui permet à n'importe qui d'authentifier un utilisateur en demandant à Yggdrasil de vérifier un jeton d'accès, n'importe qui signifie même des services externes qui n'ont rien à voir avec la société. Or dans mon système, je n'ai pas besoin que des personnes externes puissent authentifier les utilisateurs, seuls les différents systèmes que composent Webgames en ont besoin. Les JWT (qui seront expliqués dans un chapitre dédié) permettent à n'importe quel système *interne* d'authentifier un utilisateur sans devoir passer par l'autorité d'authentification. Migrer vers les JWT me fait donc gagner en performance.

Le match-maker a été un service plus difficile à rendre sans-état. Vu que l'ensemble des files d'attente étaient gérées en local, il était nécessaire de soit trouver une solution qui ne nécessite plus d'état soit de déplacer l'état en dehors de l'application. J'ai préféré la seconde solution en sauvant les files d'attente locales dans une base de données répondant exactement à mes besoins : *Redis*.

### Redis

*Redis* est une base de données nosql orientée clé-valeur (*key-value store*) stockée en mémoire vive et accessible par le réseau. Cette base de données peut sauvegarder plusieurs structures de données comme des chaines de caractères, des tableaux, des sets triés et d'autres. La force de Redis est d'être extrêmement rapide autant en lecture qu'en écriture.

Si Redis a permis de déplacer l'état en dehors du match-maker, il était nécessaire d'empêcher les accès concurrents à la base de données. Certaines opérations nécessitent d'exécuter une série d'instructions en une fois pour assurer l'intégrité des données. Après plusieurs recherches, il s'est avéré que Redis était déjà totalement capable de jouer une série d'instructions en une seule fois

mais uniquement sur un seul des noeuds d'une grappe de serveur Redis. Pour jouer un ensemble d'instructions et empêcher d'autres noeuds de la grappe d'effectuer en même temps des opérations, il faut mettre en place une logique d'exclusion mutuelle au sein de l'application; Redis n'étant pas capable de le faire seul. Malgré cette contrainte, j'ai tout de même utilisé Redis en me cantonnant à n'utiliser qu'un seul serveur.

## Communications avec le client

Si les données que manipulait l'application ont pu être déplacées en dehors de l'application, les communications réseaux ont été plus difficiles à rendre sant état.

En effet, le match-maker a besoin d'envoyer des requêtes aux clients (par exemple pour les prévenir qu'une partie débute), la communication était gérée en TCP qui est un protole avec état.

Lorsque plusieurs match-maker tournent en parallèle, les utilisateurs sont répartis parmi l'ensemble des serveurs. Chaque serveur est donc directement connecté à une partie des utilisateurs. Lorsque celui-ci désire communiquer avec eux il peut directement leur envoyer un message. Par contre il lui est impossible d'envoyer directement un message aux utilisateurs connectés aux autres serveurs.

Pour pallier à ce problème, lorsqu'un serveur doit communiquer avec un autre utilisateur, il va diffuser (*broadcast*) son message sur l'ensemble du réseau, les autres serveur vont recevoir sa diffusion et chacun pourra transmettre le message aux utilisateurs concernés.

Si cette solution a permis de rendre le système fonctionnel, il n'était plus possible d'avoir des match-makers en dehors d'un même réseau car les diffusions ne peuvent pas aller plus loin. De plus la diffusion des messages ajoutait beaucoup de trafic inutile sur le réseau et ralentissait les match-maker qui passaient leur temps à vérifier si les utilisateurs étaient directement connectés ou non. Pour finir il présentait une faille de sécurité vu que n'importe qui connecté au même réseau recevait l'ensemble des messages, même ceux ne lui étant pas destinés.

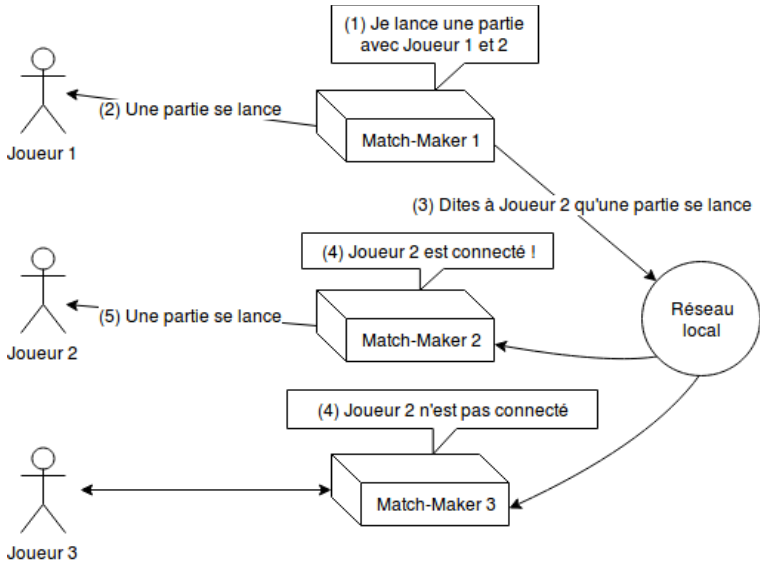


*Illustration 3: Schémas de la communication avec les clients dans Webgames v4*

## *Webgames v5 - Travail de fin d'études*

La 5e version de Webgames a pour ambitieux objectif de développer un système distribué, avec une architecture basée sur des micro-services complètement sans état, qui sera hébergé de manière élastique et utilisé par une application cliente servant d'interface utilisateur.

L'objectif est également d'avoir un environnement de développement le plus sain possible en mettant en place des tests unitaires et des tests d'intégration qui soient automatiquement exécutés à chaque mise à jour : en cas d'echec des tests, la mise à jour est annulée. Des statistiques de qualité de code et de couverture des tests seront également présents pour assurer les meilleures pratiques de développemet.

Deux environnements seront disponibles : un environnement de développement pour faire tourner l'ensemble du logiciel sans aucune dépendance externe (*standalone*) et un environnement de production qui se liera avec des bases de données dédiées.

Le projet tournera sur des containers Docker pour faciliter le déploiement. La propriété immuable des containers et le fait que toutes les dépendances sont incluses permettent de pouvoir déployer l'application à peu près n'importe où très rapidement.

Le projet sera hébergé de sorte à avoir une élasticité horizontale automatique pour à la fois être capable de supporter des pics de charge sans pour autant laisser tourner une myriade de serveurs.

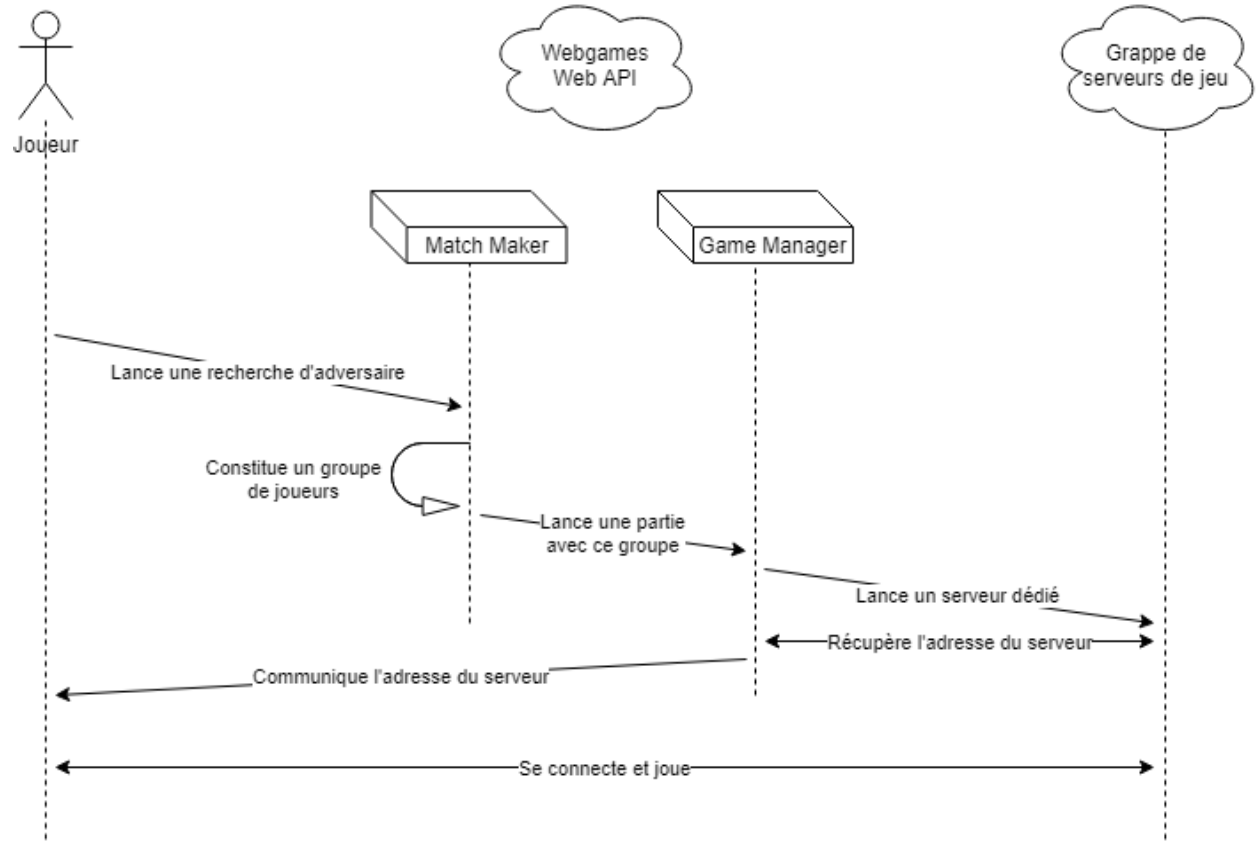


*Illustration 4: Schéma simplifié de la 5e version de Webgames*

# Sprint de janvier - Choix de l'infrastructure

> **TL; DR** Docker comme technologie de containers, Kubernetes comme orchestrateur et hébergement dans le cloud d'Amazon.

Contrairement au monde du logiciel où une application est installée chez chaque client, déployer un service est beaucoup plus compliqué:

- La sécurité des utilisateurs doit être renforcée car tout le monde utilise la même application.
- La disponibilité du service doit être assurée car en cas de bug, tout le monde est affecté.
- L'infrastructure doit pouvoir absorber une montée en charge contrairement à un logiciel qui n'est utilisé que par une seule personne à la fois.
- Les mises à jour doivent se faire sans avoir d'impact sur l'utilisateur.
- Et d'autres contraintes auxquels je ne pense pas…

Afin d'assurer les points précédents, il est important de séparer un seul gros service (*monolith*) en une série de micro-services travaillant de concert. Chaque micro-service est placé dans un *containeur* afin d'assurer son isolement, l'ensemble des containeurs est alors guidé par un *orchestrateur* qui assure leur évolutivité et leur disponibilité. L'ensemble des containeurs et des orchestrateurs est hébergé sur du matériel physique qui doit être provisionné de manière automatique en fonction de la charge sur le parc.

En résumé il est nécessaire d'emboiter trois technologies : containeur, orchestrateur et hébergeur.

## *Choix de la technologie de containeur*

Un containeur est un environnement virtuel créé au sein du système mais isolé du reste du système, une application lancée au sein d'un containeur pourra utiliser les ressources du système (ou en tout cas celles qu'on l'aura autorisées à être utilisées) mais n'aura pas accès au reste du système. Plusieurs outils se sont développées sur la technologie `cgroup` du noyau Linux qui permettent de créer ce genre de containeurs, ***Docker***, *Rocket*, *systemd-nspawn* sont les plus connus.

systemd-nspawn est le plus léger, directement inclus à systemd, il permet de créer un *namespace* complet (une réplique de la hiérarchie du système de fichier (*file system*)) isolé du reste du système. Il est possible de faire communiquer le containeur avec l'hôte en créant une interface virtuelle partagée entre les deux ou même de directement utiliser les interfaces réseau de l'hôte. Le désavantage de systemd-nspawn est qu'il ne propose aucune interface de gestion, l'administrateur système doit développer son propre logiciel de gestion autour de cette technologie.

Docker est le plus connu, utilisé par un grand nombre de compagnies, il permet en plus de l'isolement une panoplie de fonctionnalités très utiles aux DevOps ! Au moyen de la CLI (*Command Line Interface)*, il est possible de créer différents types de réseaux, de mapper certains ports du containeur à des ports de l'hôte ou encore de lier certains volumes. Toutes ces fonctionnalités sont très utiles mais pas autant que le *Dockerfile*, il s'agit d'un fichier qui va de manière déclarative décrire le contenu d'un containeur, docker va pouvoir créer une *image* sur

base de ce fichier qui pourra être utilisée comme base à d'autres Dockerfiles. Une application devient alors une image construite sur base d'une série d'ancêtres.

rkt au final est le plus récent, il permet une meilleure isolation que docker et une meilleure intégration à Linux tout en ayant les mêmes fonctionnalités et en supportant les Dockerfile. Cependant sa jeunesse fait qu'il n'est pas encore bien intégré aux outils d'orchestration.

Pour ses fonctionnalités et sa bonne intégration aux outils d'orchestrations, **docker** sera la technologie de containeur que j'utiliserai.

## *Choix de l'orchestrateur*

**Kubernetes** vs Nomad vs Mesos vs Swarm

Un orchestrateur a la difficile tâche de gérer la répartition des ressources de plusieurs machines physiques aux différents containeurs, de gérer la communication entre eux, de permettre des mises à jour tournantes et de permettre la haute-disponibilité et l'élasticité des services. Beaucoup d'orchestrateurs se partagent le marché: ***kubernetes***, *swarm*, *nomad* et *mesos* sont les plus répandus.

J'écarte d'entrée de jeu mesos et nomad pour le manque de retour que j'ai de ces technologies et plus particulièrement nomad qui n'a aucune documentation digne de ce nom.

Kubernetes est à la base un orchestrateur propriétaire développé par google sous le nom Borg mais est aujourd'hui libre et développé conjointement par google, redhat et coreos. Il a la réputation d'être stable et complet mais d'être assez difficile à prendre en main et à installer. Ses fonctionnalités puissantes en font d'ailleurs un orchestrateur très utilisé en entreprise.

Swarm est l'orchestrateur fourni de base avec docker, il évolue rapidement mais ses services limités en font plus un jouet qu'un vrai orchestrateur. Il ne propose pas à l'heure actuelle de vérifier régulièrement que l'application fonctionne et de la relancer en cas de crash.

Sans réelle concurrence de la part de Swarm et vu que je ne connais pas du tout ni nomad ni mesos, mon choix se porte sur **Kubernetes**.

## *Choix de l'hébergement*

Le cloud Amazon (AWS) propose une fonctionnalité essentielle à mon désir d'infrastructure hautement-disponible et élastique : les *auto-scaling groups*. Un auto-scaling group peut être configuré pour automa-tiquement provisionner de nouveaux serveurs lorsque la charge totale dépasse un certain seuil, il permet également l'inverse : arrêter des serveurs lorsque la charge passe en-dessous d'un autre seuil. Le sprint de mars explique plus en détail cette technologie

Le provisionnement automatique permet à l'infrastructure d'être élastique et de toujours pouvoir gérer la charge sans excès.

# Sprint de février - Développement de l'API Web

> **TL; DR** API Rest sécurisée stateless asynchrone avec injection de dépendances. Intégration continue avec des tests unitaires, des tests d'intégration, un calcul de couverture du code et un vérificateur de style.

Développer une API Rest peut aussi bien être trivial que très complexe. Accès aux données, limitation des accès, fonctionnalités proposées, application sans état, … l'API Rest n'est au travers de ses endpoints que la partie visible d'un iceberg qui peut cacher beaucoup de complexité.

## *Framework*

Écrire un serveur web "from scratch" n'est plus en pratique depuis un petit temps, des frameworks existent sous diverses formes pour la plupart des langages de programmation. Dans le monde *python* les deux plus importants frameworks sont *Django* et *Flask*. *Django* est un framework MVC, il suit la même logique de *Spring* du monde *Java*. *Flask* est un micro-framework, il ne fait que le strict minimum : parler HTTP/HTTPS et router une requête à une méthode qui va la traiter, le développeur a beaucoup plus de contrôle. Si *Django* permet de faire des applications web MVC robustes, *Flask* est plus adapté aux API RESTful.

Afin d'améliorer les performances de *Flask*, je vais utiliser ***Sanic***. *Sanic* ré-implémente la même interface que flask en y ajoutant le support pour les communications asynchrones ce qui boost ses performances au point d'être comparable à NodeJS *Express* et Go.

## *Bases de données*

Comme toute application la mienne a besoin d'enregistrer des données : des comptes utilisateurs, des jeux, des parties de jeux et d'autres. Vu la nature différente de chacune de ces données, plusieurs bases de données vont travailler de concert.

### SGBD Relationnelle

Utilisée pour les données au schéma clairement défini, je compte utiliser une *base de données relationnelle* : ***PostgreSQL***. *PostgreSQL* est un SGBD réputé, stable, puissant, propre et open-source. C'est le meilleurs SGBD gratuit disponible sur le marché et aussi celui avec lequel j'ai le plus d'expérience.

Afin de maintenir de la consistance entre différents clients et avoir toutes les requêtes SQL à un seul et même endroit : toutes les requêtes sont placées dans des procédures stockées au niveau du SGBB, utiliser des procédures permet de coder toute la logique une seule fois du côté de la base de donnée au lieu d'une multitude de fois du coté de les applications utilisant cette base de données. Ces procédures sont ensuite appelées par les clients (l'application serveur) via des *prepared statements* afin d'accélérer le traitement et de rendre impossible les injections SQL : vu que la query (ici juste l'appel de la procédure) est d'abord compilée côté SGBD, toute tentative d'injection SQL devient inopérante.

### SGBD Clé-valeur

Utilisé pour les données dynamiques avec peu de structure et pour mettre en cache des requêtes SQL récurrentes, je compte utiliser un *key-value store* : ***Redis***. *Redis* est l'un des deux KVS matures avec *Memcached*, il est open-source, rapide et stable. Contrairement à *Memcached*, des drivers asynchrones pour *Redis* existent en Python. Redis n'écrit rien sur disque, il garde tout en mémoire dans la RAM, son but n'est pas de sauvegarder durablement des données mais plutôt de permettre à plusieurs instances de l'utiliser comme mémoire partagée à accès et consultation rapides.

Pour empêcher le chevauchement des clés, seules des données générées côté serveur (identifiant unique, timestamp, textes "hardcodés") sont utilisées pour les constituer.

## *Injection de dépendance*

Dans notre projet d'intégration *Air*, j'ai développé avec Mathieu Rousseau l'API Rest du projet. Nous avions eu comme gros problème d'être incapables de tester notre application sans pouvoir nous connecter à la base de données de production. Il s'agissait d'une erreur de conception que j'ai corrigée dans mon TFE. Grâce à de l'injection de dépendance, j'ai totalement isolé l'environnement de développement de la production. Au lieu de se connecter aux bases de données dans l'environnement de développement, mon application va lancer des substituts locaux. Postgres est ainsi remplacé par sqlite et redis par des objets python.

## *Middlewares*

Un middleware est une fonction de traitement intermédiaire, dans le cadre d'une api web, elle permet de faire du pré-traitement de requêtes HTTP et du post-traitement des réponses. Dans mon application, plusieurs middlewares existent afin de gérer beaucoup plus facilement des tâches récurrentes comme l'authentification et la validation de requêtes. Des middlewares me permettent également de convertir les erreurs renvoyées par mon application (par exemple une erreur d'intégrité au niveau de la base de données) d'une réponse générique 500 - Internal Server Error à une erreur plus précise (par exemple 400 - Bad request pour les contraintes d'intégrités violées)

## *Stateless et JWT*

> "Stateless systems are the easiests ones to distribute and scale, and therefore should be designed as such when possible." *The Hacker's Guide to Scaling Python*

La plupart des sites web sont *statefull* : lorsqu'un utilisateur atteint le site, une session est créée entre lui et le serveur web. Le site web va gérer un identifiant de session qu'il va enregistrer dans les cookies HTTP. Une session est donc maintenue entre le site web et l'utilisateur ce qui permet au site de générer des pages spécifiques à cet utilisateur. Le problème de cette architecture, même si elle est plus simple à mettre en place, est que le serveur sur lequel l'utilisateur s'est connecté est le seul à pouvoir gérer ses requêtes. S'il crash, la session de l'utilisateur est perdue.

Une application stateless ne garde aucune trace en mémoire du client, aucune session n'est créée

et maintenue. À chaque requête HTTP, le client doit envoyer au serveur tout ce dont il a besoin pour travailler. C'est ici qu'interviennent les JSON Web Tokens (JWT). Au lieu que le client doive fournir systématiquement le combo utilisateur/mot de passe et que le serveur doive faire des requêtes à la base de données pour vérifier ces informations et récupérer le compte de l'utilisateur pour pouvoir évaluer sa requête, les JWT permettent de bien meilleures performances par la génération d'un jeton d'accès qui est enregistré par le client et qui contient l'ensemble des informations dont le serveur a besoin pour le traitement des requêtes.

Un JWT est constitué de trois parties distinctes:

1) Un objet JSON qui contient les informations liées à la nature du JWT (version, type de chiffrement). Utilisé pour le protocole.
2) Un objet JSON généré par le serveur qui contient les informations de l'utilisateur (identifiant unique, nom d'utilisateur, rôle de l'utilisateur, date d'expiration du jeton d'accès). Utilisé pour vérifier la validité du jeton d'accès et récupérer les informations de l'utilisateur.
3) La signature numérique de la 2e partie afin d'empêcher l'utilisateur d'altérer les informations (changer la date d'expiration ou se donner le rôle admin).

Le client est responsable d'envoyer ce JWT à chaque requête. Une fois que le serveur l'a décodé et validé, il possède toutes les informations nécessaires au traitement de la requête sans devoir passer par la base de données.

## *Sécurité et Authentification*

Lorsqu'on fait un système d'authentification local (sans passer par une source externe via OpenID par exemple) il faut pouvoir s'assurer que l'utilisateur qui se connecte est bien l'utilisateur en question et personne d'autre. La solution la plus simple est de demander que l'utilisateur définisse un mot de passe lors de son inscription, mot de passe qui lui sera demandé à chaque tentative de connexion.

Si cette solution est la plus simple elle est également la plus risquée: mots de passe trop courts, mots de passe trop simples, vol de mot de passe, attaque par homme du milieu, fishing, accès non autorisé à la base de données, attaque par force brute, … S'assurer que le secret partagé entre l'utilisateur et le serveur reste secret est un défi.

### Token Revocation List (TRL)

Pour empêcher un utilisateur de continer à accéder à l'api après qu'il se soit déconnecté, le jeton d'accès utilisé par l'utilisateur lors de sa déconnexion est placé dans une liste de révocation. Cette liste permet d'invalider un jeton d'accès même s'il n'est pas encore expiré. À chaque requête effectée avec un jeton d'accès, le serveur vérifie que ce jeton n'est pas invalidé avant de poursuivre le traitement de la requête.

### Traffic sécurisé de bout en bout

Pour assurer une protection optimale contre les attaques de type *homme-du-milieu*, un certificat TLS signé par *Let's Encrypt* est déployé au niveau du load-balancer. La configuration du load-balancer a été étudiée pour fournir le meilleur niveau de protection.

En plus du load-balancer, l'application peut également être configurée en TLS. Pour le moment je génère des certificats auto-signés pour avoir du trafic chiffré de bout-en-bout. Il serait de bon goût que je mette en place une autorité de certification locale sur un PC déconnecté du réseau et auquel seul moi ai accès. Je pourrai alors n'autoriser au niveau du load-balancer que cette autorité locale et éviter les attaques par homme-du-milieu entre le load-balancer et l'application.

## Renforcement des mots de passe

Pour protéger les mots de passe des utilisateurs, ceux-ci ne sont pas enregistrés tels quels dans la base de données. Une empreinte du mot de passe est générée à l'inscription et est vérifiée à chaque nouvelle connexion.

Les empreintes sont générées via *scrypt*, un algorithme spécialisé dans le renforcement des mots de passe. Cet algorithme est beaucoup mieux adapté à la génération d'empreintes de mot de passe que le sont les algorithmes de génération d'empreinte générique comme *md5* ou *sha1*.

*Script* permet entre autres de définir une durée minimale pour la génération de l'empreinte, configurée à 0.1 seconde dans mon application, cette durée rend très difficile les attaques par brute-force même si l'attaquant parvient à accéder directement à la base de données. *Script* n'est pas non plus sensible aux attaques par analyse des temps de réponse du processeur *timing attack*.

## Limite des tentatives d'authentification

Pour limiter l'impact d'une attaque par force brute, mon application logue chaque tentative infructueuse de connexion avec le tag `SECURITY` et l'adresse IP de la source. Il est prévu que les fichiers de log soient surveillés par *Fail2Ban*, cet outil peut être configuré pour suivre certains messages (ici ceux tagués `SECURITY`) sur une fenêtre de temps, s'il y a un dépassement d'un seuil toléré de tentatives, l'outil prend une mesure appropriée, la mesure serait dans un premier temps de limiter le nombre de requêtes pas minute et dans un second temps d'empêcher toute connexion provenant de cette adresse pour un certain temps.

## Atténuation des attaques de type DoS

Pour limiter l'impact d'une attaque par déni de service, il est prévu de limiter le nombre de requêtes par minute au niveau du reverse-proxy *nginx*. Lorsque le seuil est atteint, le reverse-proxy répondra automatiquement avec une erreur HTTP `429 – Too Many Request`.

## *Intégration continue*

### Git


On ne présente plus git. J'utilise deux branches: master et dev.

Master contient la dernière version stable de mon application et est la version qui est déployée en production, il n'est pas possible d'envoyer des modifications directement sur cette branche.

Dev est la branche de développement qui contient la dernière version nighty, cette branche n'est jamais déployée en production car elle contient la dernière version nighty, work-in-progress. Je publie sur cette branche.

Pour pouvoir publier sur master, il faut faire une pull request depuis la branche dev. La pull-request est vérifiée et validée ou non.

### Tests


Afin de valider à chaque mise à jour que tout fonctionne toujours, l'application est vérifiée par des tests unitaires et des tests d'intégration. Les tests unitaires vérifient que chaque fonction prise à part fonctionne comme prévu quelle que soit sa mise en condition. Les tests d'intégrations vérifient que l'ensemble de l'application (l'interaction de plusieurs fonctions entre elles ou avec les bases de données) fonctionnent dans les conditions normales d'utilisation.

### Vérificateur de style


En plus de vérifier que l'application est fonctionnelle, un test de qualité globale est exécuté. La syntaxe, le respect des conventions et l'utilisation des bonnes pratiques sont vérifiés et notés sur 10.

L'ensemble du projet est ainsi analysé, testé et vérifié.

## pyup


Maintenir à jour l'ensemble des dépendances et s'assurer qu'aucune mise à jour n'inclut de nouveau bug est nécessaire mais consomme beaucoup de temps, il faut se tenir informé sur le développement de chaque dépendance, installer et vérifier que chaque mise à jour est fonctionnelle. Heureusement un outil open-source "pyup" peut être installé et intégré à github. Cet outil monitor les dépendances python existantes et envoie une pull request pour chaque mise à jour disponible.

## travis


Avant qu'une série de commit puisse être envoyée en production, il est essentiel de faire tourner l'ensemble des outils cité ci-dessus.

1) Exécuter les tests unitaires et les tests d'intégration pour s'assurer que la mise à jour n'ajoute aucun bug.
2) Vérifier la couverture des tests et la qualité du code pour s'assurer que les bonnes pratiques sont en place.

Ensuite, une fois que tout est vérifié:

1) Créer le paquet qui contient l'application et l'ensemble de ses dépendances.
2) Lancer une rolling-update pour remplacer l'ancienne version actuellement en production par la nouvelle.

En réalité, je n'effectue aucune de ces opérations à la main car travis peut le faire automatiquement. Travis est un outil d'intégration continue qui s'intègre à github : il effectue l'ensemble des quatre étapes décrites automatiquement à chaque pull-request vers la branche master. Il empêche d'intégrer les changements tant que les tests ne sont pas effectués et n'autorise l'intégration que s'ils ont réussi. De cette manière aussi bien mes mises à jour (pull-request depuis la branche dev) que les pull-request proposées par pyup sont vérifiées.

# Sprint de mars - Infrastructure

**TL; DR** pivot d'infrastructure, hébergement chez moi sur du matériel récupéré de mon stage car l'hébergement sur AWS est hors budget.

## *Infra @AWS*

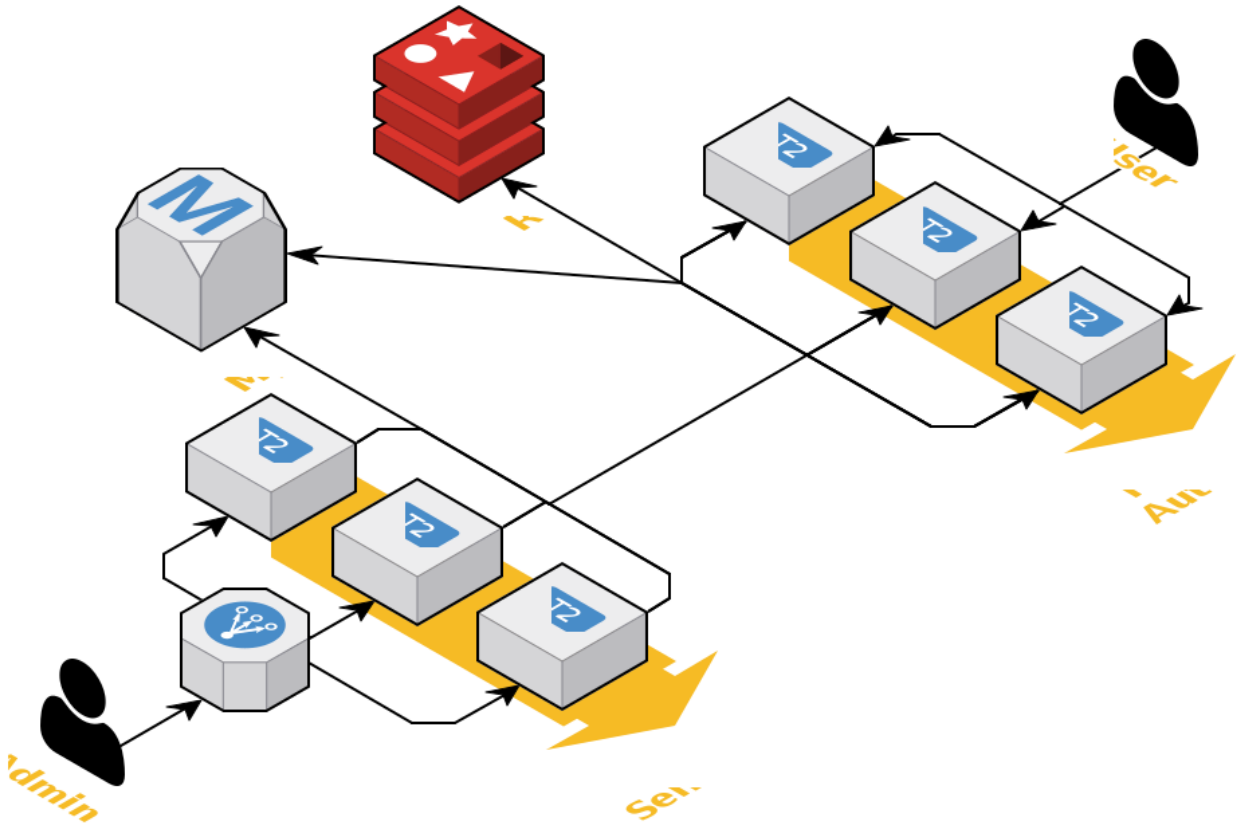

*Illustration 5: Infrastructure de Webgames au sain d'AWS*

Si Kubernetes permet la haute disponibilité et l'elasticité des *logiciels*, il ne lui est pas possible de gérer la haute disponibilité et l'elasticité de l'*infrastructure*. À ce niveau c'est AWS qui est responsable du provisionnement automatique de nouvelles machines.

Comme présenté dans le rapport de janvier, Amazon propose une solution de mise à échelle et de récupération automatique. Pour un service standalone quatre services doivent être configurés:

- ***Auto-scaling group***. Noyau de la solution, un auto-scaling group est responsable de l'élasticité et de la récupération du service. C'est lui qui lance des nouvelles instances, en arrête et remplace celles marquées *unhealthy* (malade) par de nouvelles instances. Un ASG peut être configuré pour maintenir un même nombre d'instances *healthy* dans le cluster ou être configuré pour lancer et arrêter dynamiquement des instances en fonction

de la charge du cluster et de règles définies par les administrateurs.

- ***Launch configuration***. Nécessaire au démarrage d'une nouvelle instance par l'ASG, la launch config précise l'AMI (*Amazon Machine Image*) à utiliser, les *group security* à octroyer, les *VPC*s (*Virtual Private Network*) où l'instance peut être hébergée, la taille des disques à monter et une série d'autres options.
- ***Elastic Load Balancer***. Répartisseur de la charge, un ASG aura en amont un ELB qui sera l'unique serveur visible sur internet, le seul ayant une adresse IP publique. Il recevra en entrée la totalité du trafic lié au service et répartira équitablement les requêtes vers les instances de l'ASG.
- ***Target groups***. Bourreau du service, il effectue des *healthchecks* (test de santé) sur chaque instance à interval régulier. C'est lui qui vérifie que chaque instance est toujours opérationnelle et qui marque les instances comme *unhealthy* en cas de problème.

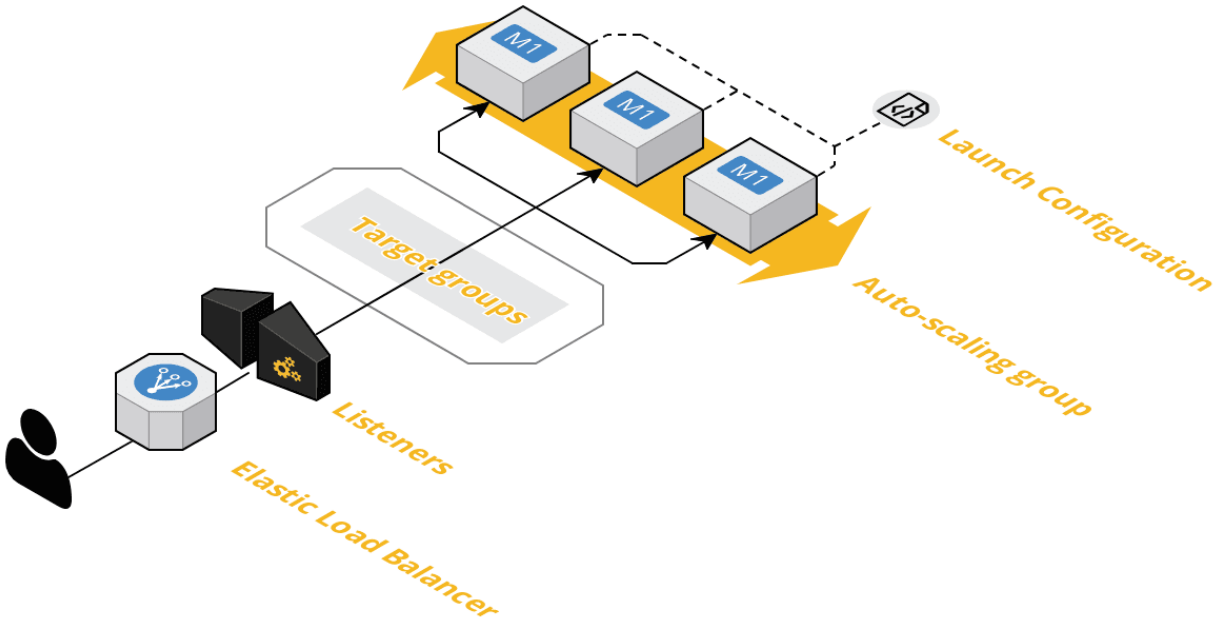


*Illustration 6: Infrastructure haute-disponibilité dans AWS*

Dans le cadre de Kubernetes seuls les deux premier nous intéresse vu qu'il est déjà capable de s'occuper des deux suivants. Pour rendre dynamique le cluster géré par Kubernetes il faut lancer une AMI avec tout ce que kubernetes a besoin pour fonctionner mais en plus que l'instance créée se connecte automatiquement au reste du cluster au démarrage.

Au départ j'avais prévu de créer une telle image à la main via Ansible et l'héberger en AMI via Packer mais une solution beaucoup plus simple s'est présentée: RancherOS.

## *Rancher & RancherOS*

Rancher est une solution tout-en-un pour accompagner le devops dans la gestion de son cluster. Au moyen d'une page web, rancher gère les différentes ressources au niveau des serveurs comme l'infrastructure, l'orchestrateur et les images dockers. Il s'intègre bien avec différents niveaux d'authentification comme LDAP et OpenID.

Rancher fonctionne particulièrement bien avec Kubernetes, il donne accès à l'interface web d'administration de kubernetes et un accès direct au CLI de kubelet.

RancherOS n'a rien à voir avec Rancher lui-même, il s'agit d'un OS minimaliste optimisé pour lancer des contenaires docker. Il est compatible avec Cloud-Init, ce qui signifie que toute la configuration de l'OS (hostname, network, users/groups, ssh, …) peut se faire au sein d'un fichier YAML qui sert à générer une image (un .iso ou un .img) avec toute la configuration prête. L'image ainsi générée se prête très bien à une installation en série car l'administrateur système n'a plus besoin de se connecter à la machine créée pour seulement la configurer.

## *Problème lié au coût*

Seulement rien de tout ceci n'a pu être mis en place, une fois que le schéma de l'infrastructure a été finalisé, j'ai pu calculer son prix. Avec les instances les plus petites possible pour minimiser les coûts, le budget nécessaire pour tout faire tourner est de l'ordre de 120€ par mois, budget complètement hors de mes capacités financières.

Une solution serait de laisser tomber la haute-disponibilité de rancher et de ne pas utiliser RDS mais dans ce cas héberger sur amazon n'a plus d'intérêt. J'ai donc décidé en milieu de sprint de laisser tomber AWS et d'héberger sur du matériel à ma disposition chez moi.

## *Infra @home*

Étant déjà propriétaire d'un serveur domestique et de quelques raspberry, j'ai commencé par installer Rancher sur une VM du serveur physique pour ensuite continuer sur les raspberry avec lesquels je n'ai eu que des soucis:

- Docker n'est pas supporté de base par Raspian OS, j'ai dû formater et installer Hypriot (une distribution linux pour raspberry avec docker en first-citizen class)
- Avec Hypriot, j'ai pu faire tourner docker et tenter d'installer l'agent client de rancher. De nouveau j'ai eu des problèmes car rancher n'arrivait pas à démarrer.
- Au final j'ai re-formatté chaque raspberry pour installer RancherOS, la distribution linux officielle de rancher. Si l'installation a bien fonctionné, rancher n'a de nouveau pas pu être lancé.

Le problème est en réalité l'architecture des CPU utilisés par les rasberry : ARM. Si cette architecture permet une bien moindre consommation d'électricité que les architecture i386 et AMD-64 pour des performences similaires, elle le fait au moyen d'un jeu d'instructions différent des cartes Intel ou AMD ce qui empêche l'agent rancher de s'exécuter.

Vu que mon stage se passait bien, mon maitre de stage a décidé de m'offrir un serveur qu'ils déclassaient. J'étais donc reparti pour toute une phase de formatage de disque, d'installation d'un hyperviseur et de provisionnement de machines virtuelles.

Au final je n'ai eu l'infrastructure minimale pour héberger mon projet (rancher + kubernetes) et mon logiciel live qu'à la toute fin du sprint de mars. La configuration de kubernetes est minimale : pas d'auto-scalling ni de self-recovery ni de monitoring ni d'health-checks et son intégration avec mon upstream nginx est inexistante. Ce sprint a été un fiasco et j'aurais aimé avoir plus de temps pour prendre en main rancher et kubernetes.

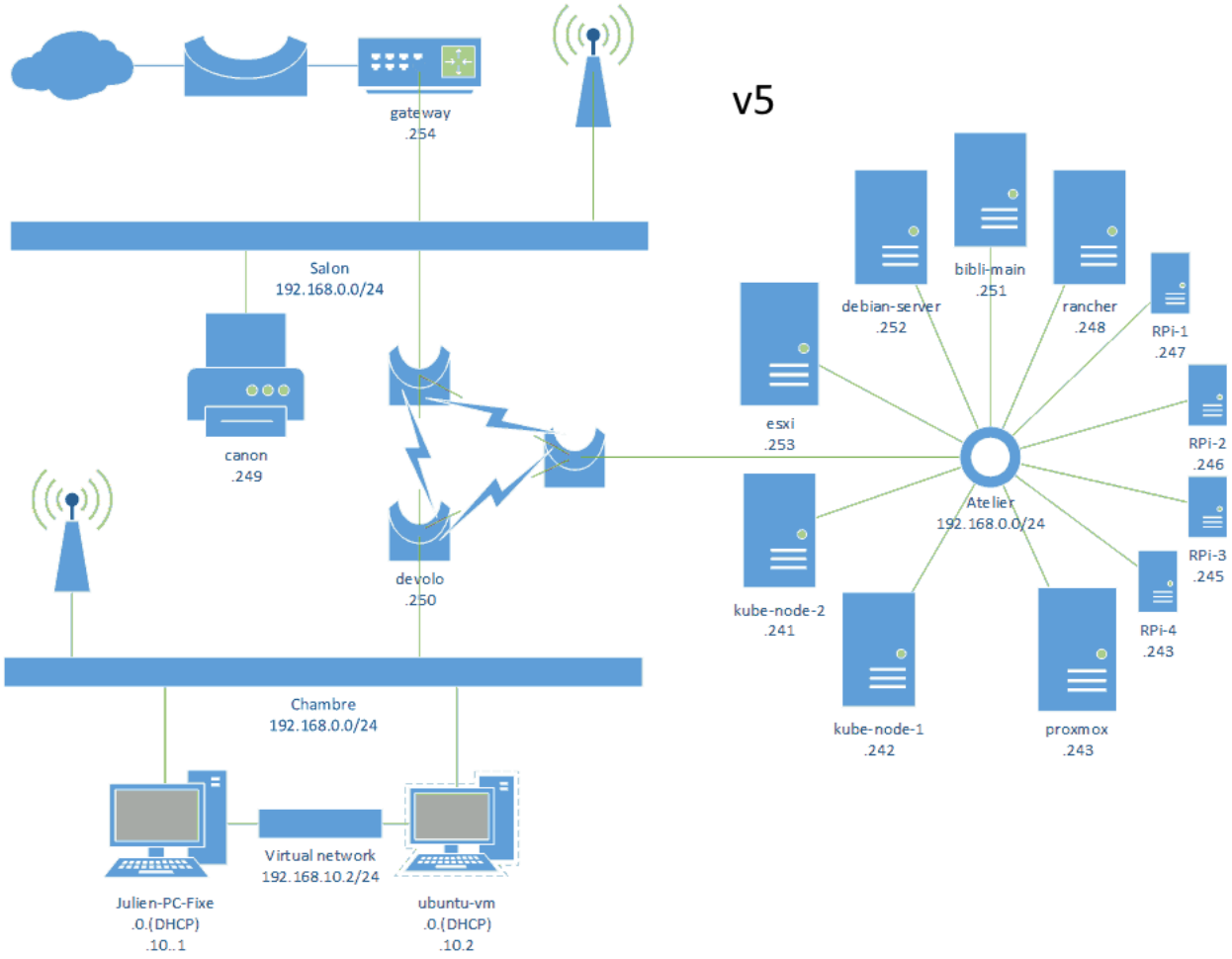


*Illustration 7: Réseau domestique*

# Sprint d’avril - Gestion des groupes et Match-Maker

Le Match-Maker est le service responsable de la gestion des groupes de joueurs et de l’algo visant à faire correspondre des groupes pour lancer des parties.

## *Gestion des groupes*

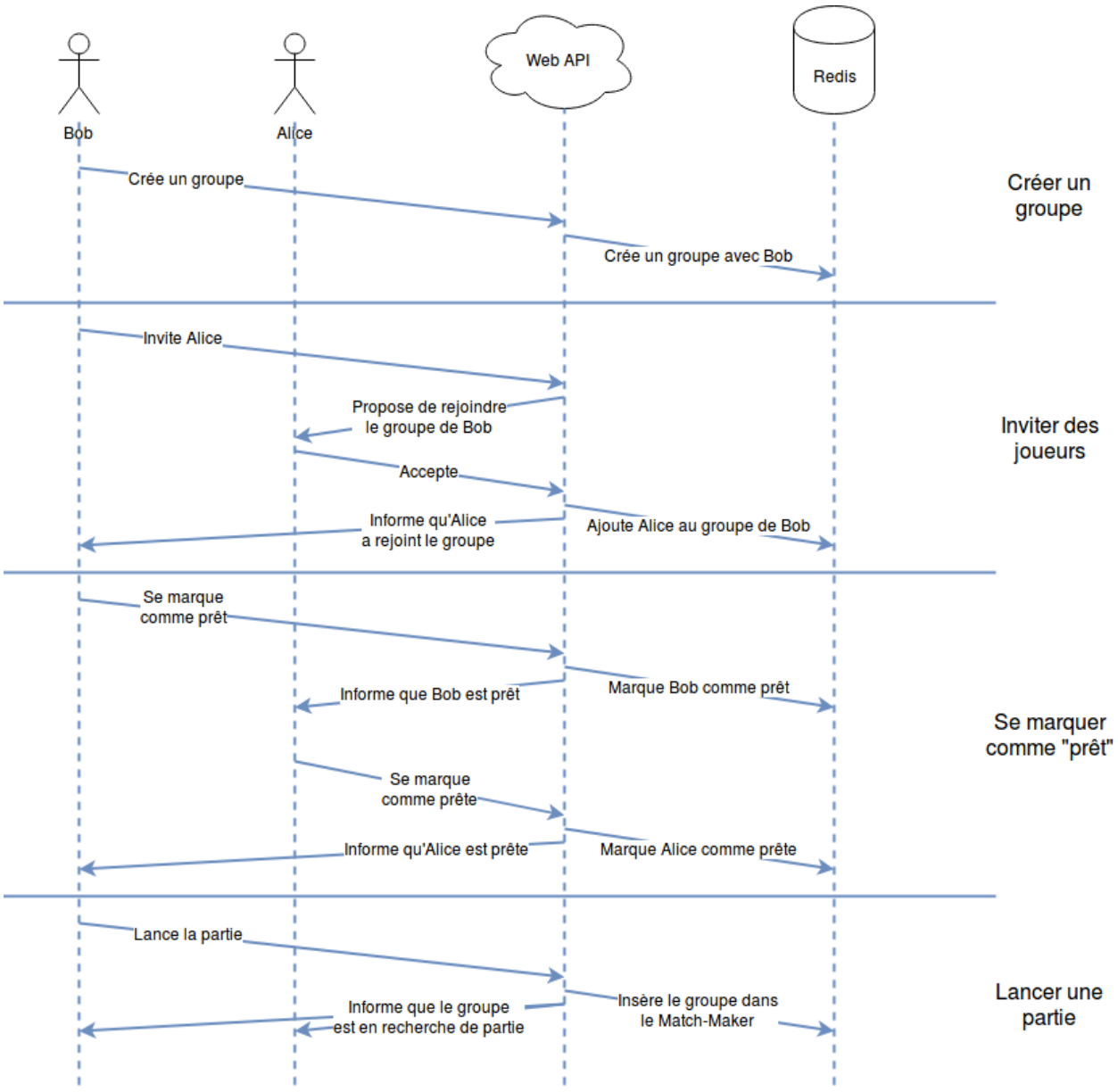


*Illustration 8: Diagrammes de la gestion d’un groupe*

Le système repose entièrement sur le *groupe* : avant de pouvoir chercher une partie, chaque joueur doit soit créer un groupe soit en rejoindre un. Il peut aussi décider de créer un groupe à lui seul et directement lancer la recherche de partie.

Chaque membre du groupe peut inviter d'autres joueurs, ceux-ci reçoivent une invitation qu'ils peuvent accepter ou décliner. Il est prévu de n'autoriser que le créateur du groupe à inviter d'autres joueurs et également de lui permettre de donner des droits d'invitation à d'autres joueurs.

Pour lancer une partie, il faut que chaque membre du groupe se soit marqué comme prêt. Cette petite limitation permet de réguler le moment où le groupe recherchera une partie laissant ainsi le temps au groupe de se constituer.

Une fois que le groupe est constitué et que tous ses membres sont prêts, n'importe qui peut lancer la recherche de partie. Encore une fois il est prévu de ne laisser que le créateur du groupe de lancer la recherche et également lui permettre de donner une dérogation à d'autres membres.

### *Service Statefull*

Du fait que le système repose principalement sur l'état du groupe de joueurs (groupe en train de se constituer, en attente d'une partie ou en train de jouer) il est *statefull*. Si pour un système classique être statefull ne pose aucun problème, il en est autrement pour un système distribué. L'état doit être partagé entre chaque noeud et les accès (lecture + écriture) doivent être mutuellement exclusifs.

Le groupe fait partie des données que je qualifie de volatiles, elles sont très dynamiques et n'ont pas besoin d'être sauvegardées sur disque. Les sauvegarder en mémoire suffit puisqu'en cas de perte des données, les joueurs pourront toujours reconstituer leur groupe. Comme expliqué plus haut, j'utilise une base de données Redis pour ce type de donnée. En plus d'être extrêmement rapide autant en lecture qu'en écriture, la base de données nous assure que *toutes les requêtes sont exécutées séquentiellement* ce qui assure l'exclusivité de l'état. De cette manière, il est raisonnable de considérer Redis comme de la mémoire partagée avec mutex accessible par le réseau.

### *Double implémentation*

Malgré que Redis soit tout droit désigné pour stocker les groupes de joueurs, la manipulation de la base de données est très laborieuse : manipulation des clés identifiant des ressources, conversion des types et utilisation de l'API de Redis pour Python font que le code est plus long et plus à même de planter.

Pour me permettre d'avoir un prototype rapidement fonctionnel, j'ai donc implémenté la gestion des groupes et le match-maker directement en python avec l'état sauvegardé en mémoire. Cette approche m'a permis de pouvoir explorer plusieurs idées dans des itérations très courtes (parfois plusieurs sur une même soirée) pour me fixer sur une solution viable que j'ai intégralement implémentée et testée.

Équipé d'une batterie de tests unitaires, j'ai pu alors m'attaquer à l'implémentation pour Redis.

L’implémenter une seconde fois m’a fait penser à des cas de figure auxquels je n’avais pas pensé dans la première implémentation. Des nouvelles fonctionnalités ont été ajoutées et d’autres améliorées, toujours accompagnées par des tests unitaires.

Cette double implémentation entre aussi dans ma volonté de séparer les tests unitaires des tests d’intégration en ayant une version standalone et une version distribuée. Avec du recul, cette approche est aussi un premier pas vers du Test-Driven-Development car au final j’avais tous les tests avant l’implémentation pour Redis.

### *Tests unitaires avancés*

Si tester une application synchrone est aussi simple que d’appeler une fonction avec certains arguments et vérifier que les valeurs de retour (ou un état) correspondent à ce qu’on attendait, tester une application asynchrone nécessite de faire jouer chaque fonction de manière synchrone. Le code des tests s’en retrouve plus long et donc plus sujet au erreurs d’implémentation.

Par exemple, au moyen de la bibliothèque `unittest` fournie de base avec Python, si des objets servant à moquer des fonctions ou des méthodes existent, aucun n’est capable de moquer des fonctions/méthodes asynchrones. Heureusement des bibliothèques externes commes `asynctest` palient à ce problème dans mes tests en ajoutant ces objets manquant.

# Sprint de mai - Application cliente et Communcations

Développer un service backend c'est bien mais il est inutile tant qu'il n'est pas utilisé par un programme client servant d'interface à l'utilisateur.

## *`urwid`, une interface client pour terminaux*

`urwid` est une bibliothèque Python pour créer des interfaces utilisateur dans le terminal, elle repose sur la puissante bibliothèque C `curses`. Je l'utilise à défaut d'être à l'aise avec quoi que ce soit d'autres, je n'ai jamais travaillé avec les bibliothèques populaires comme `Tkinter`, `wx` ou `qt` et je ne me sentais pas d'attaque pour apprendre à les utiliser. De plus je trouve que les applications terminales ont un certain charme.

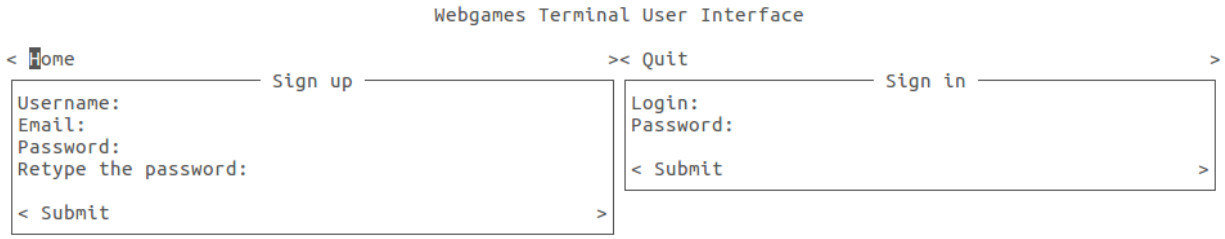


*Illustration 9: Interface Utilisateur*

### Moteur asynchrone

Si le visuel peut paraitre austère au premier regard, l'interface est en fait vraiment ergonomique et très intuitive. De plus la bibliothèque propose différents moteurs de gestion événementielle parmi lesquels `asyncio`. Le fait d'utiliser un moteur asynchrone rend l'interface beaucoup plus fluide que son équivalent synchrone : lorsque l'application fait des requêtes HTTP vers l'API, l'application ne reste pas bloquée le temps de recevoir une réponse mais rend la main à l'interface et reviendra traiter la requête HTTP lorsqu'elle sera prête.

### Interface MVC

Si dans l'ancienne version de WebGames (v4 réalisée durant les vacances d'été 2017) je découvrais la bibliothèque et avait tout écrit dans un même fichier, rendant le code très compliqué et très peu propre, pour la v5 (travail de fin d'études) j'ai choisi de directement baser mon interface sur une architecture MVC :

- Le modèle regroupe toutes les fonctions d'accès à l'API Web.
- La vue regroupe l'ensemble des composants de l'interface (boutons, champs texte, …) et leur agencement (en-tête, navigation, corps, pied de page).
- Le contrôleur est tout le code qui va réagir autant à des actions utilisateurs qu'à des événements joués par le modèle.

Travailler en MVC apporte une structure bienvenue au code. Le code est beaucoup plus concis, mieux agencé et me permet de m'y retrouver plus facilement.

## *Échange de données avec le client*

Toute la communication avec le serveur se fait via l'API Rest grâce à un client HTTP. J'ai choisi d'utiliser `aiohttp` qui est le client HTTP asynchrone de référence pour Python. Cette bibliothèque possède tout ce qu'on attend d'un client HTTP : keep-alive tcp, compabilité HTTPS et plusieurs connexions parallèles sont les plus appréciables.

Si communiquer du client vers le serveur et récupérer sa réponse est simple, l'inverse est plus ardue. J'ai également besoin que mon serveur puisse envoyer des messages à l'intention du client, si cette petite phrase semble anodine, il est en réalité très compliqué de communiquer avec un client se trouvant souvant derrière une connexion natée et un pare-feu.

### Serveur côté client

La solution la plus simple est de lancer un serveur HTTP également du côté du client, que le serveur puisse également envoyer des requêtes vers le client. Le problème est que nos clients ici sont des joueurs, généralement chez eux, sur un réseau privé derrière de la NAT. Pour accéder à un serveur sur leur machine, il est nécessaire de faire du port forwading au niveau du routeur. Si des technologies récentes comme UPnP permettent d'ouvrir dynamiquement des ports de la sorte, la technologie n'est pas encore totalement déployée sur les réseaux domestiques et rend donc cette approche très compliquée dans sa mise en oeuvre. Pour finir les pare-feux rendent la tâche encore plus ardue ce qui m'a poussé à analyser d'autres solutions qui me semblaient plus viables.

### Protocole maison

Une autre solution est de laisser tomber HTTP au profit d'un protocole maison basé sur TCP. Le client et le serveur peuvent alors utiliser une même connexion laissée ouverte (contrairement à HTTP où la connexion se ferme une fois la réponse transmise) et s'échanger des messages. Cette solution a été explorée dans la v4, si la communication a été possible de chaque côté, le fait d'écrire à la main un protocole de couche 7 n'était pas simple et j'ai préféré explorer d'autres solutions en v5.

### Polling HTTP

Une autre solution simple est le polling HTTP, à intervalle régulier le client envoie une requête au serveur pour demander si des nouveaux messages doivent lui être transmis, le serveur peut alors répondre avec tous les messages accumulés depuis la dernière requête. Si cette solution fonctionne, elle présente plusieurs désavantages :

- Le client doit envoyer une requête pour recevoir les messages au lieu de les recevoir directement ajoutant du trafic superflu.
- Le serveur répondra régulièrement rien du tout (HTTP Response Code 204) car il n'aura rien eu à dire depuis la dernière requête.
- Au lieu de recevoir les messages instantanément, le client doit attendre chaque réponse pour les recevoir.

Définir la fréquence à laquelle le client enverra une requête devient donc important, une fréquence élevée permettra au client d'être réactif mais chargera d'avantage le serveur, à

l’inverse une fréquence faible utilisera moins de ressource mais le client sera moins réactif.

Cette fréquence devrait être déterminée avec des outils de monitoring afin de trouver une qui ne surcharge pas trop le serveur et qui est acceptable par le client. En attendant d’avoir ces mesures, je m’étais fixé à une requête toute les cinq secondes.

### Streaming HTTP

Le polling HTTP était une solution fonctionnelle dont je n’étais pas du tout satisfait. Certes plus simple qu’un protocole maison, la charge réseau supplémentaire me déplaisait beaucoup. Heureusement, au fil de mes recherches, j’ai trouvé une super solution pour que le serveur puisse transmettre des infos au client : le streaming HTTP.

Le streaming HTTP est en fait une technique très répandue pour l’envoi de fichiers en ligne : vidéos YouTube, musiques soundcloud, fichiers Dropbox. Lorsque le serveur web répond avec du streaming, le client web laisse la connexion ouverte, le serveur peut alors envoyer des données n’importe quand.

Certains comportements par défaut sont par contre à prendre en compte. Certains clients web ferment automatiquement la connexion au bout d’un certain temps : c’est le cas d’aiohttp (client web utilisé dans l’interface client) qui fermait la connexion automatiquement au bout de cinq minutes. D’autres ferment la connexion si rien n’a été envoyé depuis un certain temps. Pour le premier cas, j’ai simplement configuré le client pour qu’il laisse la connexion ouverte. Pour le second cas j’ai configuré le serveur pour qu’il envoie à intervalle régulier un message spécial qui est ignoré par le client, cette technique est connue comme battement de coeur (*heartbeat*).

Cette solution répond parfaitement à mes besoins car elle résout tous les problèmes des autres solutions envisagées et les nouveaux problèmes sont entièrement gérés. Premièrement comme le client initialise la connexion il n’y a aucun problème de NAT. Deuxièmement l’utilisation du protocole HTTP à la place d’un protocole maison facilite énormément les communications et sera beaucoup plus simple à réimplémenter dans d’autres langages. Troisièmement, il n’y a pas de surcharge réseau démesurée, le battement de coeur ne nécessite que quelques octets à intervalle faible.

## *Échange de données de n’importe quel service vers le client*

Si le streaming HTTP permet à un service de communiquer avec un client, qu’en est-il des autres services ? S’il n’y avait qu’un seul serveur de streaming, la solution serait très simple : envoyer une requête HTTP vers ce serveur demandant de transmettre un message à un client précis ou un groupe de clients mais il n’y a pas un seul serveur de streaming. Comme l’architecture est distribuée et évolutive, il peut y avoir autant de serveurs de streaming qu’il n’en faut pour tenir la charge utilisateur. Le problème devient donc non trivial et a également suivi plusieurs pistes d’études.

### Diffusion

La solution la plus simple est de diffuser (*broadcast*) sur l’ensemble du réseau un segment UDP contenant la liste des utilisateurs finaux à atteindre et le message à envoyer. Cette solution a été

explorée pendant la v4. C'est solution présente énormément de désavantages, premièrement elle nécessite à nouveau l'écriture d'un protocole de communication de couche 7, deuxièmement la diffusion sur le réseau surcharge celui-ci et troisièmement chaque serveur doit comparer la liste d'utilisateurs transmise avec la liste des utilisateurs directement connectés pour transmettre ou non le message.

En raison des nombreux désavantages, cette solution a été complètement abandonnée avec la v5.

## Publication et Souscription

Le pattern de communication est en fait de la communication d'un serveur à plusieurs autres sans besoin de réponse (*one-to-many push and forget pattern*). En faisant des recherches, je me suis rendu compte que ce pattern est également connu comme publication/souscription. N'importe qui peut publier un message qui sera transmis à tous ceux ayant souscrit à ce type de message. Ce pattern résout l'ensemble des problèmes de la diffusion.

Lorsqu'un client se connecte à un serveur celui-ci peut souscrire (*subscribe*) aux messages à destination de ce client. Lorsqu'un serveur veut communiquer avec ce client, il lui suffit de publier (*publish*) son message à destination du client, les serveurs ayant souscrit (et rien qu'eux) reçoivent alors le message qu'ils peuvent transmettre au client sans traitement supplémentaire.

### *Première Implémentation via Redis*

La base de données Redis déjà utilisée par différents services propose également un service de publication/souscription (*pub/sub*). Même si Redis n'est pas à la base prévu pour, cette fonctionnalité était bienvenue et j'ai décidé de l'utiliser.

L'implémentation s'est faite assez rapidement. Au diapason du reste des drivers, j'ai fait une double implémentation : une en mémoire sans passer par Redis et une en passant par Redis. La première implémentation m'a permis d'avoir quelque chose de fonctionnel, de fixer mes idées et d'écrire les tests, la seconde m'a permis de tirer toute la puissance de Redis et de profiter des tests déjà écrits.

Malheureusement, un comportement très désagréable a été constaté au niveau du driver Python. Du fait d'une mauvaise conception, le driver n'est pas capable de souscrire plusieurs fois aux mêmes publications. Le bug est connu des développeurs depuis Janvier 2018 et une correction est en cours mais le comportement était suffisamment désagréable pour que je supprime l'implémentation sur Redis et que je cherche des alternatives.

### *Seconde Implémentation via ZeroMQ*

Zero Message Queue est une bibliothèque réseau très puissante écrite en C et qui propose des interfaces (*bindings*) vers une foison de langages dont Python. Contrairement à d'autres bibliothèques de messagerie comme AMQP ou Kafka, ZMQ n'est pas un courtier (*broker*). Les courtiers fonctionnent sur un service dédié qui sert d'intermédiaire à la transmission des messages, ils peuvent ainsi stocker les messages le temps que l'intermédiaire soit connecté et opérationnel. À l'inverse ZeroMQ n'est pas un courtier, il ne fait qu'envoyer *tout de suite* les messages sans se préoccuper que le destinataire soit connecté. Si aujourd'hui avec l'expérience je me rends compte qu'un courtier serait mieux pour mon système, mes maigres connaissances du domaine il y a quelques mois m'ont fait partir dans la direction de ZeroMQ.

ZMQ est basé sur quelques patterns (Requête/Réponse (*REQ/REP*), Envoi/Reception (*PUSH/PULL*), Publication/Souscription (*PUB/SUB*), Routage (*Route*)), il est possible de constituer un réseau de messagerie complexe et performant. Je n'ai fait qu'effleurer les possibilités de la bibliothèque dans la v5 mais je compte bien la réutiliser dans d'autres projets.

Vu que je n'avais plus le temps d'étudier ZMQ en profondeur pour faire un système de routage complexe avec de la découverte de service (*service discovery*) afin d'inclure directement toute la logique de messagerie dans le serveur, j'ai utilisé un serveur supplémentaire dédié et centralisé qui sert de d'intermédiaire à tous les autres serveurs. Ce serveur est donc un noeud de défaillance unique (*single point of failure*) dans mon architecture.

Pour qu'un serveur applicatif puisse transmettre un message à un client, il envoie (*push*) au serveur de messagerie qui la reçoit (*pull*), le serveur va à son tour publier (*pub*) le message, pour finir un serveur d'application ayant souscrit (*sub*) aux messages envoyés à ce client le recevra et pourra le transmettre au client par le flux HTTP ouvert avec ce dernier.

## *Communication inter-services*

Comme chaque service propose une API accessible en HTTP, la communication entre les différents services est triviale. Chaque service peut interroger les autres services en utilisant leur API. Pour les services soumis à un contrôle d'accès, chaque service est capable de générer un jeton d'accès avec les permissions d'administrateur.

### *Communications tenaces*

L'ensemble des communications que ce soit les communications entre le client et les services, les communications entre les services ou les communications vers les serveurs externes (redis, postgresql, serveur intermédiaire de messagerie) est tenace. Par tenace j'entends qu'une perte de connexion ne causera pas l'arrêt d'aucun logiciel.

Toutes les requêtes aux services en HTTP sont réalisées avec la bibliothèque Python `tenacity`, cette bibliothèque permet de réexécuter une fonction si celle-ci lève une exception. La configuration de la bibliothèque est comme suit et se base sur les recommandations de développeurs travaillant quotidiennement sur des systèmes distribués :

J'ai configuré `tenacity` afin de…

- relancer la fonction si l'exception est de type `ClientConnectorError`. Il s'agit de l'erreur levée par le client HTTP asynchrone `aiohttp` en cas de problème de connexion réseau. Si d'autres erreurs peuvent être levées par la fonction (comme par exemple une erreur http 500 - internal server error), elles ne sont pas propices à une réexécution de la requête HTTP.
- attendre au départ 3 secondes avant de relancer la requête et évoluer petit à petit vers un délai de 10 secondes. L'attente minimum permet de ne pas stresser le réseau avec un flot continu de requête et le délai maximum permet de ne pas atteindre des délais faramineux. Une composante exponentielle permet d'espacer de plus en plus les requêtes et une composante aléatoire permet d'éviter d'avoir un tonnerre de requêtes simultanées.

Concernant les bases de données externes, une logique de reconnexion en cas d'erreur est déjà présente de base à la fois dans `asyncpg` (driver postgresql) et dans `aioredis` (driver redis). C'est également une des raisons pour lesquelles j'ai choisi ces drivers.

Pour finir ZeroMQ, de par sa conception et son protocole, ne sera jamais impacté par une perte de connexion réseau. Si le correspondant n'est pas connecté, l'envoi est simplement annulé et le message perdu. Si ce comportement n'est pas vraiment souhaitable dans mon application, il a l'avantage de ne pas la faire planter en cas de perte de connexion avec l'intermédiaire.

## Sprint de juin - Lancement des jeux

Le lancement des jeux constitue la dernière étape de mon projet, étape qui a dû être revue du fait que l'infrastructure sur Kubernetes n'est pas prête pour déployer directement des jeux. À l'heure de l'écriture de ces dernière lignes, l'implémentation du lancement des jeux est encore en cours de développement. J'ai déjà réussi à faire ce que je voulais dans un bac à sable (*sandbox*) mais il me reste encore beaucoup de travail pour intégrer le code au reste des services de Webgames.

### *Hébergement des jeux*

Comme présenté dans le sprint de février, Docker est une technologie très répandue de container. Bien que l'utilisation de cette technologie fasse que la communication entre le client et l'application tournant dans le container soit un tout petit peu plus lente que si elle tournait directement sur l'hôte, les containers docker restent un très bon choix pour lancer à la demande des serveurs de jeu. Docker a également l'immense avantage d'être intégralement contrôlable avec une API Rest là où systemd-nspawn et rkt proposent des interfaces de plus bas niveau.

Une alternative aux containers peut-être mieux adaptée aux jeux-vidéo seraient des unikernel. Technologie innovante, un unikernel est en fait le paquetage (*packaging*) d'une application en natif sur un kernel qui a été épuré de tous les systèmes n'étant pas nécessaires au fonctionnement de l'application. Les pilotes, gestionnaires des tâches, communications réseau, etc d'un système d'exploitation classique sont à cet effet sélectionnés pour créer le système le plus léger possible. L'immense avantage est que des Unikernels sont beaucoup plus rapides à démarrer (quelques millisecondes) que des systèmes d'exploitation traditionnels (plusieurs dizaines de secondes) et à exécuter du fait de l'environnement minimaliste.

### *Lancement des jeux*

Une fois une partie constituée, l'API lance un container docker et fait correspondre les ports internes utilisés par le serveur de jeu sur des ports libres de l'hôte. L'adresse et les ports de l'hôte sont communiqués aux joueurs qui peuvent ainsi se connecter.

Afin de permettre au serveur de jeu d'identifier les joueurs, l'API génère des nouveaux JWT qu'elle signe avec un secret qui est communiqué au serveur de jeu via une variable d'environnement. Les nouveaux JWT sont ainsi communiqués aux clients qui peuvent les utiliser pour s'identifier auprès du serveur de jeu de manière sûre. Il est à noter qu'il est impossible pour Webgames de forcer la communication du JWT par les clients de manière sécurisée au serveur.

### *Fin de partie*

Lorsque le serveur de jeu s'arrête, Webgames assume que la partie est terminée et arrête la session de jeu. Les joueurs récupèrent la main sur le menu et peuvent démarrer une nouvelle partie en joignant à nouveau le match-maker.

## Diagramme final

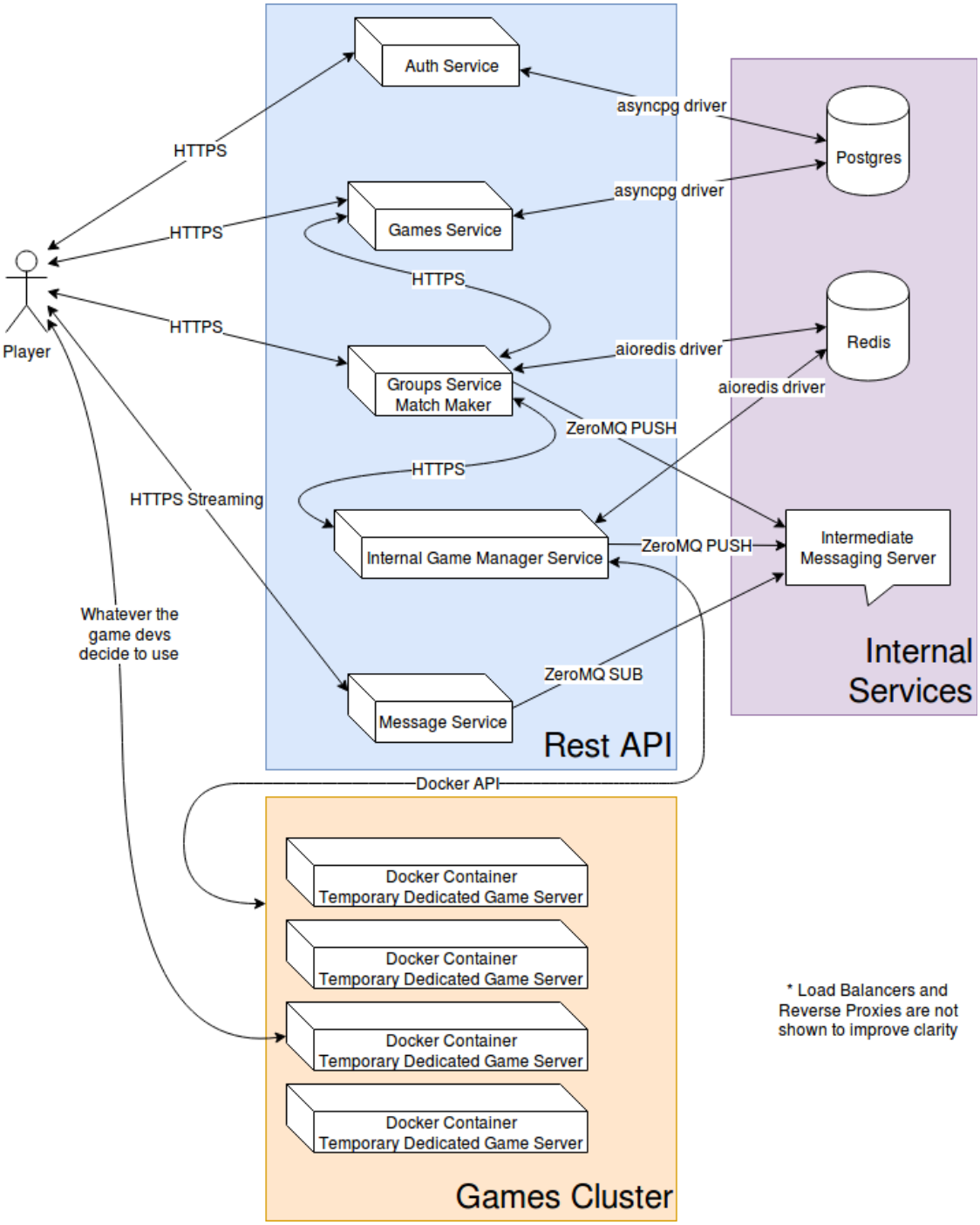


*Illustration 10: Webgames v5 - Diagramme final*

# Conclusion

Webgames v5 est la première version qui propose un prototype fonctionnel qui pourrait être présenté à des développeurs, futurs partenaires potentiels.

Bien que le système peut encore être amélioré sur tous les angles, je suis très fier du résultat final. Batir un système entièrement évolutif a été un défi que je suis heureux d'avoir accompli, il est l'accomplissement de plusieurs années de recherches et d'expérimentations comme en témoignent les quatre précédentes versions du logiciel.

L'ampleur du projet m'a fait aussi me rendre compte de mes propres limites. Le sprint de mars concernant l'hébergement doit être revu en profondeur et j'ai été très imbu de moi-même de penser que j'aurais été capable seul de tout mettre en place en un mois uniquement. J'ai également pris conscience que travailler la journée en entreprise était plus dur que je ne le pensais, nombreux sont les soirs où je n'ai pas eu le courage de travailler sur mon travail de fin d'études à la suite d'une longue journée de boulot.

Pour terminer sur une note positive, mettre en place d'entrée de jeu tout un pipeline d'intégration continue m'a énormément aidé à développer un système propre. Les mesures que me donnaient les outils de statistiques m'ont permis d'améliorer au quotidien le code de mon application et les tests automatisés ont permis de corriger de nombreux petits soucis et d'améliorer ma vélocité de développement.

# Bibliographie


[1]

Thomas H. Ptacek, « (Updated) Cryptographic Right Answers », *Gist*. [En ligne]. Disponible sur: https://gist.github.com/tqbf/be58d2d39690c3b366ad.

[2]

Nick Ma, « 5 Keys to Running Workloads Resiliently with Docker and Rancher - Part 3 », *Rancher Labs*, 00:48 - -0700-700. [En ligne]. Disponible sur: https://rancher.com/5-keys-to-running-workloads-resiliently-with-docker-and-rancher-part-3/.

[3]

Nick Ma, « 5 Keys to Running Workloads Resiliently with Rancher and Docker – Part 1 | Rancher Labs ». [En ligne]. Disponible sur: https://rancher.com/5-keys-running-workloads-resiliently-rancher-docker-part-1/.

[4]

Nick Ma, « 5 Keys to Running Workloads Resiliently with Rancher and Docker – Part 2 », *Rancher Labs*, 00:43 - -0700-700. [En ligne]. Disponible sur: https://rancher.com/5-keys-running-workloads-resiliently-rancher-docker-part-2/.

[5]

LinkedIn, « A Brief History of Scaling LinkedIn ». [En ligne]. Disponible sur: https://engineering.linkedin.com/architecture/brief-history-scaling-linkedin.

[6]

Rancher Labs, « Amazon ECS on RancherOS ». [En ligne]. Disponible sur: https://rancher.com/docs/os/v1.1/en/amazon-ecs/.

[7]

online, « amazon web services - How to do auto scaling for Rancher and Kubernetes clusters on AWS EC2? », *Stack Overflow*. [En ligne]. Disponible sur: https://stackoverflow.com/questions/47505690/how-to-do-auto-scaling-for-rancher-and-kubernetes-clusters-on-aws-ec2.

[8]

Mojang, « Authentication - wiki.vg ». [En ligne]. Disponible sur: http://wiki.vg/Authentication.

[9]

KUOKA Yusuke, *autoscaler: Autoscaling components for Kubernetes*. Kubernetes, 2018.

[10]

B. Nguyen, *awesome-scalability: High Scalability, High Availability, High Stability, High Performance, and High Intelligence Back-End Design Patterns*. 2018.

[11]

Nick Ma, « AWS and Rancher: Building a Resilient Stack », *Rancher Labs*, 00:24 - -0700-700. [En ligne]. Disponible sur: https://rancher.com/aws-rancher-building-resilient-stack/.

[12]

Google, « Configure Liveness and Readiness Probes - Kubernetes ». [En ligne]. Disponible sur:

https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/.

[13]

C. Dwork, N. Lynch, et L. Stockmeyer, « Consensus in the presence of partial synchrony », *Journal of the ACM*, vol. 35, n° 2, p. 288-323, avr. 1988.

[14]

Kristopher Sandoval, « Defining Stateful vs Stateless Web Services | Nordic APIs | », *Nordic APIs*, 11-mai-2017. [En ligne]. Disponible sur: https://nordicapis.com/defining-stateful-vs-stateless-web-services/.

[15]

Rancher Labs, « Deploying Rancher from the AWS Marketplace », *Rancher Labs*, 05:23 - -0700-700. [En ligne]. Disponible sur: https://rancher.com/deploying-rancher-from-the-aws-marketplace/.

[16]

angstwad, *docker_ubuntu*. .

[17]

Alexey Migutsky, *going-to-production: A reference checklist for topics which should be covered before going to production*. MTDV, 2018.

[18]

J. DANJOU, *HACKER'S GUIDE TO SCALING PYTHON*. S.l.: LULU COM, 2017.

[19]

Edward Angert et Sergey Pariev, « How to Install a Redis Server on Ubuntu or Debian 8 », *Linode Guides & Tutorials*. [En ligne]. Disponible sur: https://www.linode.com/docs/databases/redis/how-to-install-a-redis-server-on-ubuntu-or-debian8/.

[20]

M. J. Fischer, N. A. Lynch, et M. S. Paterson, « Impossibility of distributed consensus with one faulty process », *Journal of the ACM*, vol. 32, n° 2, p. 374-382, avr. 1985.

[21]

Thomas Auffredou, « Introduction à Terraform », *Blog Xebia - Expertise Technologique & Méthodes Agiles*, 26-janv-2015. .

[22]

C. Cachin, R. Guerraoui, et L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[23]

D. Sanche, « Kubernetes 101: Pods, Nodes, Containers, and Clusters », *Medium*, 02-janv-2018. .

[24]

Xavier Biseul, « L'unikernel : étape ultime de la virtualisation ? » [En ligne]. Disponible sur: https://www.journaldunet.com/solutions/cloud-computing/1176927-unikernel-l-etape-ultime-de-la-virtualisation/.

[25]

C. Gray et D. Cheriton, « Leases: an efficient fault-tolerant mechanism for distributed file cache consistency », 1989, p. 202-210.

[26]

Rancher Labs, « Load-Balancing in Kubernetes », *Rancher Labs*, 00:22 - -0700-700. [En ligne]. Disponible sur: https://rancher.com/load-balancing-in-kubernetes/.

[27]

John Engelman, « Managing Container Clusters with Terraform and Rancher », *Rancher Labs*, 50:36 - -0700-700. [En ligne]. Disponible sur: https://rancher.com/managing-container-clusters-terraform-rancher/.

[28]

Mathieu Agopian, « Mathieu Agopian : Python et asyncio : la recette du bonheur ? » [En ligne]. Disponible sur: http://mathieu.agopian.info/blog/python-et-asyncio-la-recette-du-bonheur.html.

[29]

codesheppard, *packer-catalog*. codesheppard, 2018.

[30]

Sergey I., Dmytro I., Dmitri I., et Sergey G., « Parallel Coding: From 90x Performance Loss To 2x Improvement », *IT Hare on Soft.ware*, 04-avr-2018. [En ligne]. Disponible sur: http://ithare.com/parallel-coding-from-90x-performance-loss-to-2x-improvement/.

[31]

Sergey I., Dmytro I., Dmitri I., et Sergey G., « Parallel STL for Newbies: Reduce and Independent Modifications », *IT Hare on Soft.ware*, 26-avr-2018. [En ligne]. Disponible sur: http://ithare.com/parallel-programming-for-parallel-noobs-reduce-and-independent-modifications/.

[32]

D. M. Beazley et B. K. Jones, *Python cookbook*, Third edition. Sebastopol, CA: O'Reilly, 2013.

[33]

A. Martelli, *Python en concentré*. Paris: O'Reilly, 2007.

[34]

R. Chowdhury, *python3-openid: Python 3 port of the python-openid library*. 2018.

[35]

Rancher Labs, « Quick Start Guide », *Rancher Labs*. [En ligne]. Disponible sur: /docs/rancher/v2.x/en/quick-start-guide/.

[36]

Nick Ma, « Resilient Workloads with Docker and Rancher - Part 4 | Rancher Labs ». [En ligne]. Disponible sur: https://rancher.com/resilient-workloads-docker-rancher-part-4/.

[37]

Nick Ma, « Resilient Workloads with Docker and Rancher: Part 5 | Rancher Labs ». [En ligne]. Disponible sur: https://rancher.com/resilient-workloads-with-docker-and-rancher-part-5/.

[38]

Alex Andrews, « Scrum Of One: How to Bring Scrum into your One-Person Operation », *Ray Wenderlich*. .

[39]
ACISSI, *Sécurité informatique: ethical hacking, apprendre l'attaque pour mieux se défendre*. 2017.

[40]
Google, « Services - Kubernetes ». [En ligne]. Disponible sur: https://kubernetes.io/docs/concepts/services-networking/service/.

[41]
M. Laccetti, *terraform-aws-rancher-ha: Terraform module for standing up an AWS Rancher HA instance*. 2018.

[42]
B. Maxwell, *terraform-rancher*. 2018.

[43]
codesheppard, *terraform-rancher-starter-template*. codesheppard, 2018.

[44]
Chetan Giridhar, « Understanding Python GIL », *CallHub*, 30-juin-2016. .

[45]
SYSADMIN BADASS, « Une infrastructure scalable, capable de résister à d'importants pics de charge - OVH ». [En ligne]. Disponible sur: https://www.ovh.com/fr/news/usercase/une-infrastructure-scalable-capable-de-resister-a-d-important-pics-de-charge.xml.

[46]
Amir Chaudhry et Richard Mortier, « Unikernels, meet Docker! | Unikernels ». [En ligne]. Disponible sur: http://unikernel.org/blog/2015/unikernels-meet-docker.

[47]
T. D. Chandra et S. Toueg, « Unreliable failure detectors for reliable distributed systems », *Journal of the ACM*, vol. 43, n° 2, p. 225-267, mars 1996.

[48]
Jon, « User Data for automated RancherOS instances », *Obviate.io*, 27-avr-2016. [En ligne]. Disponible sur: https://obviate.io/2016/04/27/user-data-for-automated-rancheros-instances/.