Rapport de stage

BUT Informatique

Par Alexandre Meunier

22 janvier - 30 mars (10 semaines)

Adresse IUT : 99 avenue Jean Baptiste Clément 93430 Villetaneuse
Tuteur enseignant : Claire Toure
Développement de handlers d'API pour le logiciel OpenSVC

OpenSVC

Adresse entreprise : 13 rue de la baronne James de Rothschild 60270 Gouvieux

Maitre de stage : Christophe Varoqui
Tuteur de stage : Cyril Galibern

Remerciements

Remerciements pour mon maître de stage Christophe Varoqui et mon tuteur de stage Cyril Galibern pour leur patience, leur travail, l'accompagnement et les connaissances qu'ils m'ont apportées.

Remerciement à ma tutrice de l'IUT, Claire Toure pour son aide à la construction du rapport de stage et pour être à l'écoute.
Merci à Pascale Hellegouarch pour tout son dévouement pour nous permettre de trouver un stage, ses conseils, et ses cours très utiles.
Merci aussi à Haïfa Zargayouna soucieuse de la réussite de notre stage, et de son écoute auprès des étudiants.
Et merci à Bouchaïb Lemaire pour ses cours sur la qualité de développement qui m'ont beaucoup aidé dans ce stage.

Remerciement à Marco Faustini, ingénieur système embarqué à Viveris, pour m'avoir aidé à chercher un stage.
Et merci à ma famille pour m'avoir aidé à trouver un stage, surtout à mon père qui m'a donné le contact de l'entreprise OpenSVC.

Table des matières

Introduction

Je suis actuellement en deuxième année de BUT Informatique à l'IUT de Villetaneuse. Pour le premier stage en BUT, j'ai voulu choisir un stage dans la programmation de logiciel car c'est ce qui m'intéresse le plus dans tout ce qu'on a découvert en cours. J'ai ainsi trouvé ce que je voulais dans l'entreprise OpenSVC. Cette entreprise, située à Gouvieux dans l'Oise (60), travaille sur le développement d'un logiciel qui assure la haute disponibilité des services d'entreprises avec des données sensibles comme des banques ou des assurances.

Ce qui a été étonnant dans la recherche pour le stage, c'est que la première entreprise à qui j'ai envoyé un mail, c'était OpenSVC, en octobre. Je l'ai connue car mon père, ébéniste, a travaillé pour Monsieur Varoqui, dont le fils était dirigeant d'une petite entreprise en informatique, sous le nom de OpenSVC. En allant sur leur site, j'ai découvert qu'ils travaillaient sous le langage Python, ce qui était parfaitement ce que je voulais. Après une longue recherche de stage sans réponse dont deux renvois de mails vers OpenSVC, cette entreprise a enfin répondu positivement à ma requête fin décembre.

Ce stage m'a proposé le développement d'un logiciel en langage golang. La mission que l'on m'a donnée était la création de points d'entrée[1] d'API, composants en charge de préparer les données à renvoyer en réponse à une requête d'un client. Cette mission a pour but de m'introduire au développement de l'agent om3 en Go, qui est le portage en golang de l'ancienne version en Python.

Au-delà de m'apprendre le développement informatique, ce stage m'a permis de vivre pendant 10 semaines dans la peau d'un développeur logiciel dans une équipe passionnée.

[1] Dans ce rapport, le handler va être renommé par sa traduction en français, le point d'entrée.

Partie 1 : Présentation de l'entreprise

1 - Présentation du produit

OpenSVC est une entreprise qui développe, commercialise et supporte un logiciel open source. Ce logiciel est chargé de démarrer, arrêter et relocaliser les applications du client, et d'en assurer la réplication des données, afin d'en assurer la haute disponibilité. C'est-à-dire que l'agent de OpenSVC va être capable de basculer des services hébergés sur un serveur vers un autre si ce premier a un incident technique. L'agent va donc surveiller tous les évènements qui se produisent dans un cluster, un groupe de serveurs, pour assurer la haute disponibilité de leurs services.

2 - Histoire

OpenSVC a été créé en 2009 par Christophe Varoqui et Cyril Galibern, pour répondre à un besoin d'un système de cluster[2] suffisamment simple pour permettre un déploiement généralisé sur un grand nombre de serveurs.

Le nom qu'ils ont donné à leur projet dérive de "Open" pour signifier open source et "SVC", acronyme de service car le logiciel supervise des services.

Cyril et Christophe se sont rencontrés en travaillant tous deux dans le service de gestion de l'infrastructure de la Société Générale Corporate Investment Banking. C'est dans ce contexte, qu'un outillage "maison" avait été développé pour répondre au besoin de haute disponibilité à grande échelle, et que la volonté de fournir un tel logiciel en Open Source est née.
La première version OpenSVC a été produite en python. La version 1.9 a été un changement majeur pour le produit qui s'est enrichi d'un daemon[3], un élément central pour surveiller et maintenir la haute disponibilité des services en restant constamment allumé.
Pendant dix ans les versions de pythons se sont succédées, chacune introduisant des incompatibilités nécessitant des ajouts de code pour supporter les différentes versions de l'interpréteur. La volonté de gagner en performance, en fiabilité et de faciliter la maintenance et la distribution du produit ont conduit à décider de porter le produit vers un langage compilé. Le développement en golang de la troisième version de OpenSVC a donc démarré en 2021.

[2] Ensemble de serveurs
[3] Un daemon est un processus exécuté en arrière-plan, qui effectue des tâches tant qu'il n'est pas arrêté

L’entreprise a commencé à recevoir des stagiaires et alternants en 2018, dans le but d’identifier de bons candidats pour la génération suivante. A ce jour, l'entreprise à accueilli 2 stagiaires et 2 alternants. Ils espèrent que mon arrivée dans l’entreprise me donnera envie de reprendre un poste chez OpenSVC ou au moins de me conforter dans mon choix de carrière.

3 - Le type de client

Le client type de OpenSVC gère des services et des données sensibles dont la disponibilité doit être assurée. Le niveau technique des équipes de gestion de l’infrastructure nécessaire pour gérer efficacement un déploiement OpenSVC est assez élevé.

Les entreprises du secteur Banque et Finance, des Telco, et de l’administration d'État sont les utilisateurs types de ce produit.

Pour répondre à l’exigence de compétence de pointe pour gérer efficacement la haute disponibilité des services, la société propose également des formations sur les systèmes d’exploitation Unix/Linux, le stockage de données et sur le clustering.

4 - Organigramme

L’entreprise est organisée en 3 pôles d’activité :

- Recherche et Développement
- Support et formation
- Prestation de services en régie

Elle compte 6 Employés :

- Varoqui Christophe : Président, R&D, support, formation, maître de stage
- Galibern Cyril : R&D, support, formation, tuteur de stage
- Veron Arnaud :Administration système, support, formation, suivi commercial, relation client, relation humaine
- Veron Florence : Secrétariat
- Gérald et Sébastien : Prestation de service en régie

Étudiants :

- Rémi : Stagiaire de août 2023 à mi-février 2024
- Théo : Alternant depuis septembre 2023

Dans la suite de ce rapport, je citerai ces personnes par leur prénom.

4 - Les relations

Tout d'abord, j'ai eu une relation un peu différente de ce que j'aurais pensé. Comme il y a moins de 10 personnes dans l'entreprise, j'ai été en contact très proche avec la plupart d'entre eux. En effet, j'ai pu prendre connaissance pendant les premières semaines avec Christophe, Cyril, Rémi ainsi que Théo après la troisième semaine. De plus, j'ai pu tutoyer les personnes dans l'entreprise ce qui m'a donné plus confiance en moi pour discuter, en diminuant ma timidité.

La première personne avec qui j'ai communiqué et travaillé était Rémi, présent pendant les deux premières semaines de mon stage. En effet, comme il était en stage depuis 6 mois, il a acquis beaucoup de connaissances et il a pu m'expliquer et m'introduire dans le projet. J'ai pu le voir comme mon tuteur de stage pendant ces deux premières semaines car il a pris du temps pour moi. On voit donc une différence considérable entre quelqu'un qui vient d'arriver dans l'entreprise et quelqu'un qui y est depuis longtemps. Cela veut aussi dire qu'il y a une bonne évolution entre le début et la fin du stage.

Il n'y a généralement que le mardi où tout le monde se rejoint en présentiel, ce qui permet de discuter entre nous et de s'aider si on est bloqué sur quelque chose. De plus, tous les repas sont pris en charge par l'entreprise. Cela permet d'aller au restaurant et d'améliorer la relation de confiance entre nous et surtout pour que je m'intègre bien dans l'entreprise. Les discussions sont pour la plupart du temps des choses personnelles comme les vacances, la famille, mais parfois le travail. J'ai pu me décontracter pendant les repas car c'est un moment calme et amical.

Quand Rémi à fait son dernier jour, on a fêté son pot de départ dans un restaurant et comme le stage s'est bien passé, l'entreprise lui à offert une carte cadeau. Ce qui montre qu'ils ont de l'importance pour leurs stagiaires et alternants et qu'ils ont entretenu une bonne relation.
Après le départ de Rémi, j'ai rencontré Théo, qui travaille principalement avec Arnaud.

Dans l'entreprise, il n'y a donc pas vraiment de présence hiérarchique et la relation dans l'équipe est très amicale, ce qui simplifie grandement la communication.

5 - L'organisation du travail

a) Les lieux de travail

Les deux premières semaines, je n'ai été qu'en présentiel, ce qui a permis à Christophe et à Rémi de m'expliquer les missions, le produit OpenSVC, et de pouvoir m'installer un poste convenable pour travailler.

Le lieu de présentiel est un peu particulier car OpenSVC n'a pas de locaux, nous travaillons donc au domicile de Christophe, ce que j'ai trouvé étonnant au début de mon stage. Il y a aussi le fait d'être aussi proche d'un président, je n'aurai jamais pensé cela avant de trouver un stage. Cette organisation de travail évite la pression sociale et le bruit des bureaux ou des open space. L'intimité de ces conditions de travail m'a aidé à contrôler la timidité et à favoriser la concentration.

J'ai pu constater que les salariés ne faisaient pas de pause le matin ni l'après-midi. De plus, on partait manger entre 12h00 et 13h00 et je reprenais directement mon travail après le déjeuner pendant qu'ils prenaient une pause café. Je ne prenais pas de café car je n'aimais pas cela. Mais comme j'avais un peu de fatigue, je me suis mis à en prendre, et à force j'ai bien aimé et j'ai pris l'habitude.
J'ai par la suite remarqué qu'ils n'avaient pas d'horaires fixes et comme Christophe travaille chez lui, il ne perd pas de temps à se déplacer vers son lieu de travail. Ce qui est pratique quand on a beaucoup de travail à faire.

Cependant, étant dans une petite entreprise avec peu de personnel, dans mon cas, j'ai trouvé qu'on était souvent isolé dans notre travail sans trop communiquer et le seul contact que l'on avait était soit de l'aide, soit de savoir si tout allait bien. Cela dépend si on veut travailler seul ou en équipe. Mais, j'avais quand même de la compagnie avec Rémi et Théo qui venaient les mêmes jours que moi pour travailler en présentiel.

Pour ce qui est du trajet, j'ai heureusement une voiture et j'ai pu être autonome pour me déplacer en présentiel. Le lieu est très proche de chez moi, à 20 minutes en voiture, et je suis à contre-courant des bouchons du matin et du soir.

Dès que j'ai pu être autonome pour travailler sur ma mission, j'ai fait du distanciel. Ce qui est étonnant avec le distanciel c'est que cela ne m'a pas beaucoup changé de l'habitude car je travaille aussi dans une maison en présentiel. J'ai pu garder un confort et une tranquillité que je retrouve dans les deux cas. Mais en distanciel, cela est plus accentué par le fait d'être chez soi et donc de faire ce que l'on veut tant que l'on respecte les horaires et que l'on travaille, je pouvais par exemple écouter de la musique en fond sans déranger personne.

En télétravail, la communication se fait principalement en messagerie instantanée. Occasionnellement, j'ai pu passer des appels vocaux avec Christophe et mon tuteur pour qu'ils m'expliquent des choses et pour que je puisse poser des questions. J'ai pu remarquer que la communication est beaucoup plus lente par écrit que par audioconférence ou qu'en présentiel, car le message à l'écrit doit être lu par l'autre, mais l'autre n'est pas forcément attentif à ses messages. C'était plus simple de poser des questions en présentiel qu'en distanciel car je ne restais pas bloqué à attendre la réponse pour agir.

La cinquième semaine a été un peu spéciale car comme Christophe est parti en vacances, et que l'entreprise n'a pas de locaux pour travailler en étroite collaboration avec l'équipe. J'ai

passé une semaine en distanciel. Heureusement, j'avais mon tuteur et Arnaud Veron qui étaient à l'écoute et pouvaient m'aider en visioconférence si j'avais des problèmes. Cette semaine m'a ainsi montré qu'être dans une petite équipe a des avantages mais aussi des inconvénients.

Pour finir, j'ai pu faire quelques jours au sein d'un espace de coworking accompagné de mon tuteur à Paris. Ce lieu est fait pour les entreprises qui n'ont pas de locaux propres à leur activité.

La première fois que nous y sommes allés, il n'y avait pas beaucoup de personnes et donc cela était assez calme pour travailler. Mais les autres jours étaient plus "bondés" et il était plus difficile de communiquer. Ces jours m'ont permis d'installer des fonctionnalités sur mon pc avec l'aide de Cyril pour pouvoir utiliser les logiciels de l'entreprise, pour programmer et tester mon code mais aussi pour comprendre certains principes du Go.

Malheureusement, je ne vois que très peu mon tuteur, seulement le mardi et les jours à Paris en coworking. J'ai cependant mon maître de stage, Christophe, à qui je peux poser des questions en présentiel, même si je peux utiliser l'outil pour communiquer.

b) Les horaires

Au niveau des horaires, je commençais à 8h30 le matin et je finissais à 17h00. Mais je restais parfois 15 minutes ou 30 minutes de plus à travailler pour essayer de finir mes missions.

Les horaires pendant les jours de coworking changeaient un peu car comme je devais prendre le train, je n'arrivais que vers 9h00 et s' il y avait des problèmes de train je pouvais arriver après 10h00. Le soir, pour rentrer il était 17h30, 18h00 et comme j'habite loin, je n'ai pas beaucoup de trains pour rentrer chez moi, donc il faut partir en avance pour ne pas le rater. C'est un des désavantage d'habiter loin de Paris, dans un endroit très peu desservi par les transports.

Ces horaires me changent de ceux de l'IUT car on n'est pas soumis à des horaires fixes. Mais aussi, comme ce n'est pas un cours, je pouvais prendre mon temps à comprendre quelque chose, et j'avais de l'aide plus facilement.

De plus, ayant plus de temps pour moi par comparaison quand j'avais cours, cela m'a permis de me libérer en fin de journée et de prendre du temps à faire des projets personnels.

Partie 2 : L'environnement de travail

1 - La compréhension du sujet

a) La compréhension du produit

Pour comprendre mes missions, j'ai d'abord dû comprendre le fonctionnement du produit et de ses divers composants. Le code source est disponible sur GitHub en open source.

Le projet principal est om3, la troisième version majeure de l'agent OpenSVC, développée en Golang. Il se compose d'un daemon, d'un Cluster Resource Manager[4] et d'un client d'administration à distance.

La compilation du projet produit deux binaires[5]:

- om : daemon, CRM, et client administration locale
- ox : client d'administration à distance

Chaque nœud[6] d'un cluster OpenSVC exécute un daemon. Les différents daemons du cluster vont pouvoir se donner des ordres par une API REST (cf annexe 1, page 31) servie sur le protocole https[7], et échanger leurs états par émission continue d'événements par plusieurs canaux indépendants et redondants : les heartbeats[8] configurés sur le cluster (cf annexe 2, page 31).

Un cluster héberge des "objets", qui vont encapsuler les applications et les données du client. Le cluster va s'assurer que ces objets ont une instance démarrée à tout instant sur un des nœuds opérationnel. En cas de panne d'un nœud, les instances qu'il faisait tourner sont remplacées par des instances sur d'autres nœuds du cluster.

---

[4] Gestionnaire de ressources du Cluster
[5] Binaire : Un binaire en Go contient le programme et tous ses imports dans un seul fichier. Cela permet de n'installer qu'un seul fichier lors du déploiement du programme sur les serveurs, ce qui est plus simple.
[6] Dans ce projet, un noeud représente un serveur
[7] Protocole qui permet la communication entre un serveur et un client sur Internet.
[8] Battements de coeur

Une instance peut donc se définir comme le point de présence d'un objet sur un nœud. Elle peut se composer de données persistantes, d'adresse ip, et de logiciels. Les logiciels sont classiquement intégrés sous la forme de micro-containers docker[9].

Ainsi, un micro-container contient un logiciel et les librairies système dont il dépend, ce qui le rend insensible aux mises à jour du système d'exploitation qui l'exécute. Un micro-container est livré sous la forme d'un seul fichier dont la signature garantit la non-altération et l'authenticité et permet de faciliter son installation. Enfin, l'exécution d'un micro-container peut être limitée en ressources consommées : cpu[10], mémoire, réseau et disques...
On trouve typiquement sous forme de micro-containers les moteurs de base de données, les serveurs web, et les middlewares[11].

Les actions d'administration des nœuds et des instances peuvent être faites directement par l'API, ou en utilisant la commande "om" sur le nœud accédé par ssh, ou en utilisant la commande "ox" sur un poste de travail client, déporté du cluster[12]. Ainsi, "ox" permet la gestion de plusieurs clusters depuis un même poste. "om" utilise typiquement une socket Unix[13] pour dialoguer avec le daemon, alors "ox" utilise une socket Inet.

On comprend alors que la plupart des fonctionnalités du produit sollicitent à la fois un point d'entrée de l'API et une sous-commande "om" ou "ox".
Par exemple, la commande "ox node print config --node dev3n3" va récupérer par l'API le contenu du fichier de configuration du nœud dev3n3 et l'afficher dans le terminal client (cf annexe 3, page 31).

La société développe également un projet nommé "collecteur", qui permet de collecter les états et piloter l'ensemble des clusters d'un client. Un collecteur se compose d'une base de données, d'un cache Redis, d'une API REST et d'une application web en python.

La migration du collecteur en golang est également en cours, mais à la différence du projet agent cette migration peut être progressive. Le serveur web est configuré pour router des chemins d'API soit vers un serveur "oc3" en golang, soit un serveur "oc2" en uwsgi python.
Cette cohabitation a déjà permis de porter du code oc2 vers oc3 en réponse à des problèmes de performance remontés par les utilisateurs.

Cette compréhension des projets OpenSVC sont l'aboutissement d'un travail sur le développement de ces logiciels. J'ai pu apprendre à utiliser et comprendre ces différents outils au gré de missions assez variées et accessibles pour approfondir mes compétences acquises à l'IUT

---

[9] Docker : C'est un logiciel qui va gérer des conteneurs, beaucoup utilisé dans le développement de logiciel avec des conteneurs.
[10] Processeur
[11] Middleware : Point de passage quand le client se connecte pour lui donner une direction à prendre
[12] Déporté du cluster signifie un ordinateur personnel utilisé pour se connecter sur des noeuds en ssh
[13] En réseau, une socket est une sorte de tuyau que va utiliser un process pour parler avec un autre. Une Socket Unix est une socket qui permet d'échanger des données par l'intermédiaire d'un fichier. L'API du daemon OpenSVC utilise à la fois une socket unix et des socket Inet, pour aller sur Internet.

b) Les objectifs de mes missions

I) Mission 1 : api om3 de découverte des noeuds

Au début de mon stage on m'a confié une mission pour me familiariser avec le projet om3. Cette première mission consistait à développer un lot de points d'entrée d'API et les sous-commandes d'administration correspondantes dans les commandes ox et om.

Ces points d'entrée ont pour but de recevoir, stocker dans un cache fichier, et restituer les informations de nœud : version de système d'exploitation, caractéristiques de cpu, mémoire, disques.

Après avoir appris à manipuler l'accès distant par ssh, le système d'exploitation Linux et l'environnement graphique de développement goland, j'ai pu commencer à programmer.
Tout au long de cette mission, j'ai appris beaucoup de choses comme le fait de décrire des points d'entrée d'API dans le manifeste au format OpenAPI 3.0 (cf annexe 4, page 32), d'implémenter ces points d'entrée, ou de savoir utiliser la commande curl (cf annexe 5, page 33) sur bash pour pouvoir tester ces points d'entrée en envoyant des requêtes et en traitant la réponse sous format JSON[14].

Cette première mission a été particulièrement laborieuse en raison du grand volume de données à décrire dans le manifeste de l'API et sous forme de structure en golang.

De plus, j'ai appris à créer des commandes avec la librairie Cobra (cf annexe 6, page 34), très populaire dans l'écosystème Golang. Elle permet de créer des sous-commandes et de décrire les options supportées par chacune.
J'ai pris un peu plus de 3 semaines pour finir cette première mission. Il restait alors environ 7 semaines de stage. Mon maître de stage estimait que je la finirai vers la moitié de mon stage si tout se passait bien.

Ce premier développement a été choisi pour son isolation fonctionnelle dans le projet. Il ne demandait que peu de connaissances sur le projet. Cela a donc été une mission pour m'initier dans le développement d'un logiciel et pour apprendre à utiliser tous les outils pour travailler.

II) Mission 2 : alimentation oc3 de workers

Pour la suite de mon stage, on m'a donné pour mission de développer la chaîne de réception dans le tout nouveau collecteur oc3 des données manipulées dans mon premier développement om3. Cette chaîne se compose d'un point d'entrée d'API qui pousse les

[14] JSON : format pour représenter les données de façon structuré

données reçues dans une file d'attente tenue par Redis, et d'un "worker", c'est-à-dire un daemon qui dépile les messages dans cette queue pour leur appliquer un traitement long : le dispatch des données en base de données relationnelle.
Pour cette mission, je devais donc d'abord comprendre le fonctionnement de la base clé-valeur Redis et utiliser les acquis de ma première mission pour le développement du point d'entrée d'API oc3.

Point d'intérêt particulier : la file d'attente Redis se comporte en FIFO[15], notion vue dans le cours de programmation système.

Pour procéder au développement, j'ai dû comprendre le code du collecteur Python pour avoir une idée de son fonctionnement et pouvoir porter le code en Golang. De plus, J'ai pu utiliser les compétences scolaires en base de données en exécutant des commandes triviales comme un "SELECT" pour vérifier que mon code peuplait[16] bien la BDD relationnelle[17].

J'ai aussi eu l'occasion de manipuler les micro-containers pour accéder aux BDD mariadb[18] et Redis.

III) Mission 3 : api om3 d'inventaire des drivers supportés

La fonctionnalité de Cluster Resource Manager de la commande "om" repose sur un système de drivers. Un objet étant composé de ressources comme une adresse IP, un système de fichier, un micro-container, … le démarrage d'une instance de cet objet nécessite que le CRM sache activer chacune de ces ressources. C'est donc au drivers d'implémenter ces méthodes spécifiques de démarrage et d'arrêt.

Dans ce contexte, il est important que l'utilisateur puisse consulter la liste des drivers disponibles d'un agent OpenSVC qui est exécuté sur un nœud. L'API de l'agent met cette liste à disposition.

La mission consistait à développer ce point d'entrée, qui sert les données stockées dans un cache fichier rafraîchit au démarrage du daemon.
Cette mission rapide utilisait tous les acquis de la première mission. Elle a permis de valider ces acquis.

[15] First In First Out, premier arrivé premier sorti
[16] Équivalent de stocker des données
[17] Base de données contenant des tables.
[18] Système de gestion de Base de données

IV) Mission 4 : parallélisation des requêtes api om3

Dans une de mes missions, j'ai appris à manipuler le parallélisme en Go. Ce principe consiste à faire avancer plusieurs fichiers d'instructions du programme en même temps. Dans le langage Go, on va utiliser des goroutines.

Cela est similaire à un fork entre deux processus[19], qui est le fait de dupliquer un processus pour donner un père et un fils. A partir du fork, le processus père continuera son fil d'exécution indépendamment de son fils.
Les goroutines en go sont comme des processus qui exécutent les lignes de code en même temps, mais au sein d'un même processus du système. Se référer au 7 de l'annexe (Page 34) pour avoir un exemple de ce que j'ai pu faire.

V) Mission 5 : vers plus d'autonomie

Par la suite, ayant une meilleure compréhension du projet, j'ai pû chercher par moi-même des opportunités de corrections et d'améliorations. En testant différentes sous-commandes "ox" et en cherchant dans le code des incohérences, j'ai pu soulever des problèmes intéressants à traiter.

C'est souvent en utilisant cette méthode que mon maître et mon tuteur identifient les chantiers à mener. Ils n'ont pas une liste exhaustive de tâches à faire, mais plutôt des problèmes qu'ils trouvent un peu par hasard en utilisant le produit pour leurs propres besoins ou à l'occasion du traitement d'un autre dossier.

Par exemple, durant mon stage, nous avons constaté que certaines séquences de démarrage et d'arrêt du daemon dans un contexte particulier pouvaient conduire à un double-démarrage du daemon sur le même nœud. Ce qui pose problème car il peut y avoir des conflits entre eux. Il fallait donc comprendre comment fonctionnaient ces chemins de code afin de les améliorer. J'ai ainsi appris que les processus Unix ont un état, et qu'on peut utiliser des signaux pour manipuler cet état.

Les différentes missions m'ont ainsi permis de comprendre la structure du code, de quelle manière il faut prendre les problèmes et la démarche intellectuelle de comprendre finement le problème avant de chercher à le corriger.

c) Les difficultés pour comprendre le contexte du sujet

Pendant la première semaine de stage, je me suis initié au Go avec le site go.dev/tour. On retrouve dans ce tutoriel, des cours sur les bases du langage avec des exercices, qui étaient

[19] Un processus en informatique est un programme qui exécute une file d'instructions

très pratiques pour apprendre. De plus, j'ai pu trouver des similitudes avec le langage C, comme par exemple le typage strict des variables et le minimalisme des mots clés réservés. J'ai aussi pu retrouver des exercices similaires à ce que j'ai pu faire en cours de développement efficace comme l'exercice sur le parcours de chaque branche et racine d'un arbre (cf annexe 8, page 36). J'ai pu facilement comprendre ce que demandait l'exercice, mais il a fallu le faire en Go, ce qui rendait l'exercice plus difficile. Comme je n'y arrivais pas, j'ai demandé de l'aide à Rémi et mon maître de stage. On n'a pu réussir qu'après avoir regardé le code source de la librairie de binary tree, ce qui nous a permis de comprendre le bon usage de cette librairie. Ce qu'on avait fait tenait sur 15 lignes, la solution tenait sur 6 lignes. Dès qu'on a trouvé la solution, on s'est rendu compte que c'était très simple et même trivial, il n'y avait pas besoin de faire plein de manipulation pour le résoudre. J'ai ainsi pu apprendre que quand le code commence à rendre quelque chose d'illisible visuellement, c'est qu'on peut réduire ou refaire quelque chose de plus propre. J'ai aussi pu voir que le golang permet très facilement d'accéder au code des librairies utilisées par nos programmes et que cet accès permet souvent de mieux en comprendre le fonctionnement et d'améliorer nos usages de ces librairies.

Pour continuer, l'accompagnement initial de Rémi m'a permis d'apprendre beaucoup de choses sur GitHub et Golang. On a aussi configuré mon poste pour qu'il soit prêt pour effectuer la première mission. Je pouvais aussi compter sur Christophe quand Rémi n'était pas en présentiel pour m'expliquer des termes que je ne connaissais pas, comme le wildcard qui est une des premières choses qu'il m'a apprise. C'est ce qui permet d'insérer une valeur dans un "print". Ces wildcards sont souvent utilisés dans les messages d'erreurs pour afficher une valeur en particulier (cf annexe 9, page 38). Cependant, quand il m'expliquait quelque chose, les termes utilisés étaient très techniques et il rentrait beaucoup dans les détails, ce qui m'a habitué à me concentrer. Chacune de ses explications étaient pertinentes et même si j'avais du mal à suivre, je repartais toujours le soir avec une somme de connaissances considérable.
De plus, j'avais à chaque fois beaucoup de mal à comprendre ce que je devais faire, comme au tout début avec les points d'entrée d'API, je n'en avais jamais entendu parlé avant. Il y avait aussi les collecteurs qui restaient pour moi quelque chose de mystérieux, mais que j'ai pu maîtriser après mes missions sur oc3.

Au niveau du temps, j'en ai passé beaucoup à comprendre ce que je devais faire. Mais cela a payé car après la première commande que j'ai créé, j'ai pu me créer une suite de fichiers à modifier, ce qui a grandement accéléré mon travail. En effet, le plus long a été la déclaration de chaque type rencontré dans le manifeste OpenAPI 3.0 de l'API. Cela demandait de comprendre la structure des données stockées dans un cache en local pour les passer en type d'API.

Du côté de oc3, pour comprendre l'algorithme pour gérer les clés de Redis, j'ai créé moi-même un algorithme à la main pour essayer de reproduire le résultat attendu.
De plus, Arnaud m'a expliqué le fonctionnement du collecteur, ce qui m'a éclaircit beaucoup de zones que je ne comprenais pas dans ma mission sur oc3.

Au final, depuis le début de mon stage et jusqu'à la fin, j'ai beaucoup progressé sur l'autonomie, la compréhension du code et j'ai pu améliorer mes compétences sur les différents langages que j'ai utilisés.

2 - L'environnement technique

a) Les outils pour construire mon environnement

Mon environnement de travail se composait de mon ordinateur portable et d'un cluster de 3 nœuds dédié à tester mes développements.
Depuis mon ordinateur personnel, je pouvais accéder à la messagerie instantanée Mattermost. Quand j'ai découvert cette messagerie je l'ai tout de suite comparée à Discord qui fait beaucoup de choses similaires. En effet, sur cet outil plus professionnel, on peut échanger des messages avec une autre personne ou dans un groupe. On peut aussi faire des appels vocaux en groupe ce qui est très pratique. En cas de besoin de visioconférence, l'entreprise met à disposition Jitsi. De plus, chaque channel de discussion est spécialisé sur chaque projet, comme celui pour om3 où j'ai le plus parlé. Mais il y a aussi des channels pour parler de tout ce qui n'a pas de rapport avec l'entreprise, un endroit pour parler en mode "décontracté".

b) Les outils pour programmer

I) Linux et son environnement pour le développement

Le système d'exploitation à été la première chose que j'ai configuré sur mon PC. Sur Linux, j'ai pu utiliser bash pour l'exécution de commande, le langage Shell pour Linux. Mais même si j'avais l'habitude de cet environnement, j'ai découvert énormément de commandes et de raccourcis claviers pour travailler plus efficacement. J'ai par exemple installé Terminator, un gestionnaire de terminaux qui permet d'ouvrir plusieurs terminaux avec la même application. Je pouvais ainsi les mettre côte à côte pour travailler sur deux terminaux en même temps, l'équivalent de deux écrans avec chacun un terminal ouvert. Ce qui est très pratique quand je devais inspecter des logs[20] et en même temps exécuter des commandes pour en voir le résultat (cf annexe 10, page 38). Cependant, il peut y avoir un inconvénient à cela, c'est qu'on peut se perdre si on en ouvre trop, ce qui nous oblige à faire un tri pour s'y retrouver.

Au niveau de l'édition de texte, j'utilisais nano, un éditeur installé de base sur Linux. Mais il est très limité sur son fonctionnement, comme le fait de ne se déplacer qu'une lettre par une

[20] Message envoyé par le daemon ou par le système d'exploitation pour donner des informations sur les différentes actions de ceux-ci

lettre. Pour avoir un éditeur plus performant, on m'a fait découvrir Vim, un éditeur de fichier beaucoup plus poussé. En effet, il contient une grande quantité de raccourcis qui permettent d'accélérer le travail. Ainsi, il existe sur Vim des raccourcis clavier comme "i" pour pouvoir éditer, "dd" pour supprimer une ligne. De plus, elles peuvent être combinées avec des nombres pour faire l'action sur plusieurs lignes comme "4p" qui permet de coller 4 fois ce qu'il y a dans le presse papier. Mais il y a aussi les raccourcis commençant par ":" comme ":x" pour sauvegarder le fichier et quitter. C'est avec tous ces raccourcis que j'ai pu modifier des fichiers en SSH avec une plus grande efficacité qu'avec nano.

II) Les langages

Pour continuer, au niveau des langages il y avait bash sous Linux mais aussi Golang qui a été la plus grosse partie de mon stage car je n'ai fait quasiment que ça. C'est un langage compilé, plutôt qu'interprété comme Python, et qui est très prisé pour sa performance, sa protection de la mémoire, sa gestion des goroutines, de ses queues de message et de la qualité de sa librairie "time". En effet, Go est réputé pour sa maîtrise du parallélisme qui permet d'avoir une flexibilité au niveau du traitement d'action faite simultanément sans chevauchement. Mais il y a aussi la gestion du temps, qui est trop souvent méprisée par beaucoup de personnes dont moi-même. On ne porte pas assez attention à ce paramètre qui peut faire basculer la bourse d'une banque à cause d'une microseconde de différence. Go ne permet pas au développeur de travailler avec des variables de temps sans zone de temps explicite, par exemple UTC. Ce qui permet d'échanger ces valeurs de temps avec des systèmes partenaires dans des zones de temps différentes sans ambiguïté. Christophe a beaucoup insisté sur la pertinence de ce langage, que j'ai trouvé très intéressant à utiliser. J'ai par exemple fait la remarque qu'il y a une importance sur le choix du type de la variable qui contient des types que je n'avais pas encore vu. Comme le type de map (cf annexe 11, page 38) qui se rapproche beaucoup de la manière dont fonctionnent les dictionnaires en python ou encore les tableaux en PHP. Cela permet aussi une optimisation en choisissant avec un type plus adapté à un autre pour une utilisation particulière. Il y a aussi le type de struct qui est une structure et va contenir plusieurs variables avec pour chacun un type différent, on peut voir cela comme un tableau mais chaque élément aura son propre type (cf annexe 12, page 39).

C'est donc intéressant de voir qu'avec un seul langage on peut faire tant de choses. Il se rapproche à la fois du C, du Python mais aussi du Java avec les interfaces, c'est ce que j' ai conclu par rapport à ce que je connaissais. J'ai pu apprendre qu'on pouvait facilement être productif en quelques semaines mais que pour maîtriser un langage il faut beaucoup d'années d'expériences, et même après 15 ans, on apprend toujours des choses.

Il y a aussi un type de format que j'ai découvert, qui est le YAML[21]. Il m'a permis de construire les structures de type pour les API. Ce langage est utilisé pour écrire des fichiers

[21] Yet Another Markup Language

de configuration et il est similaire au JSON. Ainsi, il est assez lisible pour que l'humain comprenne assez facilement.

J'ai aussi utilisé le langage SQL sous mariadb pour accéder à la BDD du collecteur oc3. J'ai pu utiliser mes connaissances que j'avais déjà sur ce langage, et j'ai appris quelques aspects en plus pour créer une table. Comme le fait de préciser le type de caractère que l'on accepte comme UTF-8 sur une colonne de la table.

III) ssh

Pendant mon stage, j'ai utilisé le ssh pour me connecter aux différents serveurs mis à ma disposition pour pouvoir tester mes programmes. J'avais déjà utilisé ce système de connexion pour un projet à l'IUT. Je connaissais donc un minimum le principe et son utilisation sécurisée. Les machines que j'utilisais, dev3n3, pour la version de om3 et infra-1, pour le collecteur, m'ont permis d'apprendre la structure que peut avoir une infrastructure en interagissant avec les daemons, les objets installés et sur le collecteur les différents conteneurs.
Il y avait aussi un aspect très utile avec le ssh, c'est que je pouvais envoyer mon code vers ces machines, ce qui permettait aux autres de pouvoir relire mes potentielles fautes dans le programme. Les développeurs de l'entreprise disposent chacun un cluster de 3 nœuds pour tester, ce qui permet d'isoler le travail de chacun sans casser ce que l'autre a fait.

III) Goland

J'ai pu utiliser comme IDE[22] Goland, édité par la société JetBrain grâce à la licence étudiants gratuite. Cette licence donne accès à de nombreux IDE très complets comme PHPStorm que j'ai pu utiliser pour développer un site pour un projet à l'IUT. Ainsi, Goland va être spécialisé dans le langage Go. On peut par exemple faire beaucoup d'actions Git sans passer par un terminal. Mais il est aussi pratique pour la gestion de sauvegarde automatique, de la recherche d'erreurs dans le code et aussi du débogage[23]. Cependant, comme il est très puissant, il est aussi très énergivore. Il me faisait donc ralentir mon PC qui n'est pas très puissant. Je pouvais aussi configurer du ssh pour se connecter à des serveurs, ce qui est très pratique pour envoyer des fichiers pour tester sur la machine en étant connecté en SSH.

[22] Integrated Development Environment, logiciel permettant une meilleure visualisation du code avec des jeux de couleurs, mais aussi de compilation pour vérifier les erreurs syntaxiques du code et tout un tas de fonctionnalités qui améliorent grandement l'efficacité du développeur.
[23] Déboguer désigne le fait d'exécuter un programme dans un certain mode qui va retourner des informations sur les différentes actions de ce programme.

IV) GitHub

Au niveau de GitHub, j'ai appris beaucoup de choses par rapport à l'IUT. J'ai été amené à faire un fork sur les projets opensvc/om3 et opensvc/oc3 dans mon espace github personnel (ameunier/om3 et ameunier/oc3), et à cloner ameunier/om3 sur mon poste de travail. J'ai dû configurer sur ce dépôt les relations "remote" avec l'upstream opensvc/om3 en plus de l'origine ameunier/om3.
J'ai pu créer des Pull Requests[24] de mes forks github vers leur origine opensvc afin de proposer la fusion de mes livrables dans le projet. Cette PR[25] contient les commits[26] prêts à être fusionnés dans le projet d'origine.

La PR est donc un bon moyen de vérifier la structure et la netteté du code. De plus, quand j'envoyais une PR, elle était passée en revue par les personnes qui gèrent le github upstream, dans le contexte du stage c'était Christophe et Cyril. Ce qui est pratique c'est qu'ils peuvent mettre des commentaires sur le code pour signaler des problèmes. C'est donc une façon pour moi de me corriger et d'apprendre de mes erreurs. C'est aussi un travail en plus pour mon maître de stage et mon tuteur de vérifier mes fautes. De plus, chaque commentaire est en anglais, et toutes les communications sur github se font dans cette langue universelle. Cela permet d'avoir un projet qui favorise les contributions internationales.

Il y a donc tout un langage et des processus GitHub qu'il faut maîtriser et comprendre. La PR est donc un outil important de la qualité des développements, mais également un outil de communication à destination des observateurs extérieurs. Les PR montrent que le projet est vivant, et témoignent du respect des valeurs open source.

J'ai des PR faites tout au long de mon projet et j'ai pu prendre l'habitude de séparer mon travail en plusieurs commits isolés pour faciliter la revue de code.

Quand on crée une nouvelle PR ou que des commits sont ajoutés à une PR existante, le projet opensvc/om3 est configuré pour exécuter les tests automatisés de qualité, notamment la commande "go test ./…".
Malheureusement, quand une PR ne passait pas car ces tests Go que GitHub effectue ont échoué, je rajoute des commits pour résoudre le problème ce qui rend la PR moins lisible. J'ai appris dans ce cas à recréer une PR en fusionnant des commits pour rétablir la meilleure lisibilité.
J'ai aussi pu apprendre un peu à utiliser Git sous forme de commande en ligne, comme voir l'état de l'espace de travail ou regarder les différences. Mais j'ai aussi pu créer un token pour l'utiliser sur Goland. Ce token permet de s'authentifier avec mon compte GitHub sans avoir à stocker mon mot de passe sur un serveur. Cette configuration permet aussi à Goland de modifier des choses sur mon espace GitHub de façon automatique.

---

[24] Sur GitHub, action de proposer les modifications du projet "forké" vers le projet d'origine
[25] Pull Request
[26] Ensemble de modifications apportées aux fichiers d'un projet (cf annexe 13, page 39).

c) Installation au poste de développement

L'installation d'un poste de développement est très important pour un développeur mais surtout chez OpenSVC car c'est tout une infrastructure qu' ii faut intégrer à notre poste. J'ai dû prendre une semaine pour finir mon installation avant de pouvoir commencer à travailler sur dev3n3.

J'ai tout d'abord installé, avec l'aide de Rémi, un système Linux en dual-boot, c'est-à-dire que je peux utiliser Linux mais aussi Windows sur le même ordinateur. On a dû réorganiser l'espace du disque dur pour ajouter Linux sur mon PC, un processus assez délicat si on ne s'est pas renseigné avant car cela peut casser les PC. Comme mon PC est assez récent, il n'était pas trop compatible avec Linux. En effet, on a dû acheter une clé USB pour la connexion au réseau sans fil qui ne fonctionnait pas, et le micro intégré ne marchait pas non plus, ce qui m'obligeait à utiliser mon casque micro en distanciel. De plus, l'écran plantait parfois, entraînant Goland avec lui, à cause de la carte graphique qui n'était pas vraiment compatible avec Linux. Nous n'avons pas pu résoudre ces problèmes, mais depuis que j'ai installé un driver spécial pour ma carte graphique, je n'ai plus rencontré le plantage de l'interface graphique.

On a aussi installé le langage Go, essentiel si je voulais travailler, mais aussi Goland. Depuis le début jusqu'à la fin du stage j'ai beaucoup modifié la configuration de cet IDE. En effet, Cyril a changé son interface pour qu'elle soit plus efficace pour programmer. J'ai installé des raccourcis pour déployer mon code vers dev3n3, en utilisant ssh. J'ai ainsi découvert beaucoup de choses qui ont optimisé mon travail. Je ne me rendais pas compte avant de la richesse des IDE, et je saurai dorénavant réutiliser ces nouvelles compétences.

On m'a aussi appris à créer des alias de commandes qui permettent d'exécuter un fichier à un certain endroit, comme l'alias goland qui ouvre directement l'IDE. j'ai aussi appris la gestion des variables d'environnement comme PATH qui contient la liste des répertoires dans lesquels chercher les programmes exécutables. Mais aussi d'autres variables qui vont influencer le comportement de mes commandes.

De plus, pour que je me connecte en ssh sur les différentes machines, on a dû me créer une paire de clés, privée protégée par mot de passe et publique, et configurer la machine distante pour qu'elle puisse m'authentifier grâce à ma clé publique. Je n'avais pas idée de ces étapes à suivre avant d'arriver à se connecter en ssh.

d) L'utilisation de ChatGPT

Avant mon stage, je n'avais jamais utilisé ChatGPT car je n'en voyais pas l'intérêt. Mais depuis que Christophe m'a conseillé, j'en ai pris l'habitude. C'est en effet un outil très puissant pour chercher des informations et pour comprendre des choses. Je l'utilisais pour qu'il m'explique un terme que je ne connaissais pas ou encore pour me donner des

raccourcis sur Goland ou sur un aspect que je ne comprenais pas en Go. Cela permettait aussi d'alléger le travail de mon maître de stage qui subissait toutes mes interrogations.
Entre temps, je suis passé sur Gemini, le produit concurrent édité par Google, mais dont la base de connaissance est plus à jour pour les utilisateurs non-payant. C'est un inconvénient qu'à chatGPT en mode gratuit car il n'est à jour que jusqu'en 2022. Ce qui est difficile de poser une question sur quelque chose qui vient d'arriver ou sur l'actualité.

La différence principale entre ces IA et les moteurs de recherche traditionnels comme celui de Google, c'est que c'est beaucoup plus rapide d'obtenir une information pertinente. Avant d'utiliser ces IA, quand j'avais un problème dans un programme, je devais aller sur plusieurs sites de FAQ comme Stackoverflow et lire beaucoup d'informations sans rapport avec mon problème.

Partie 3 : Développer dans un logiciel complexe

1) Qualité de développement

J'ai retrouvé quelques cours que j'ai eu en qualité de développement pendant mon stage. En effet, pour mon premier jour en coworking avec Cyril, il m'a montré le fonctionnement des interfaces en Go avec le principe de ségrégation de la méthode SOLID vu en cours. Ce principe consiste à séparer des besoins. Cela permet d'avoir une fonction qui implémente deux interfaces, une pour chaque besoin, au lieu d'une seule interface satisfaisant les deux besoins mais qui est systématiquement partiellement utilisée.
J'ai aussi pu découvrir que Go pouvait avoir des types publics mais aussi privés. Comme en Java, les types publics peuvent être utilisés à l'extérieur du package où il se trouve et les types privés ne peuvent être utilisés que depuis ce package (cf annexe 14, page 39).

J'ai appris qu'il valait mieux commencer par n'utiliser que des variables et types privés, pour ensuite, si nécessaire, créer des fonctions publiques pour enrober les fonctions privées à exposer, ou pour lire ou modifier les variables privées à exposer. Cette bonne pratique améliore la sécurité et facilite la maintenance du package, en application de la méthode Open de SOLID (cf annexe 15, page 39).

2) Les tests

Développer des tests a été une bonne partie de mon stage surtout vers la fin, ce qui était assez compliqué. J'ai pu mettre en œuvre les tests pour vérifier si une valeur était pareille qu'une référence ou vérifier qu'on reçoit bien une erreur, ce que j'avais vu en cours de Qualité de développement.

J'ai pu me rendre compte que ce n'est vraiment pas simple de faire des jeux de tests, car dans le cas de OpenSVC, il y a beaucoup de scénarios à tester. D'autant plus que l'agent om3 est un code assez complexe à tester à cause de la présence d'un daemon. Il faut penser ce qui pourrait se produire dans les grandes lignes, ce qui est le plus efficace pour toucher le plus grand panel de possibilités. De plus, j'ai pu découvrir par moi-même, que quand on introduit du code qui modifie le fonctionnement du système, cela fait souvent échouer des tests, et ces problèmes sont parfois très long à résoudre. Surtout qu'à mon niveau sur la connaissance du projet mais aussi sur la programmation, je ne savais pas trop quoi faire quand il y avait une erreur dans le test. Mais j'ai aussi appris que quand un test ne fonctionne pas, même tout petit, le problème doit être analysé et résolu au plus tôt, et ne pas le laisser sur le côté au risque de perdre l'utilité de tous les tests, ce qui pourrait avoir un impact important plus tard sur le projet. Quand le test échoue alors qu'avant il fonctionnait, soit on a fait une régression, soit c'est simplement le test qui doit s'adapter au programme.

De plus, j'ai pu apprendre le concept de "mocker" qui est très intéressant dans les tests quand on veut contrôler ce que retourne une fonction. On peut par exemple faire en sorte qu'une fonction retournant une chaîne de caractères, quand elle est mocké dans le test, renvoie une valeur particulière. j'ai trouvé cela très intéressant pour les jeux de tests même si je n'ai pas vraiment eu l'occasion de l'utiliser.

Je ne m'attendais pas à ce qu'on puisse faire autant de choses dans les tests. On peut par exemple préparer des fichiers de configurations de nœud ou d'objets OpenSVC dédiés au tests, des jeux de données de référence copiés à partir de résultats de requêtes à l'API. Cela permet au test d'avoir une base pour laquelle s'appuyer sans modifier les fichiers de configuration d'origine ou d'avoir des fichiers de configuration différents d'un test à l'autre.

J'ai aussi pu découvrir un risque lié aux tests. C'est que parfois ce test à besoin de modifier le code source du programme principal, ce qui peut le complexifier ou le rendre moins performant (cf annexe 16, page 40). Dans la plupart de ces cas, on peut utiliser le mocking éviter ces modifications du code source.

3) La gestion des erreurs

J'ai appris dans ce stage que la gestion des erreurs est très importante pour avoir un code fluide, lisible et compréhensible peu importe le cas d'erreur. En effet, quand on reçoit une erreur renvoyée par le programme, il doit nous dire clairement ce qu'il ne va pas, car un message d'erreur peu compréhensible peut faire perdre beaucoup de temps à essayer de tourner autour de l'erreur en cherchant ce qui pourrait être associé.

De plus, dans le code, il ne faudrait pas sauter une erreur peu importante car on peut sous-estimer son importance. C'est souvent de petites erreurs, comme un caractère en trop, qui peuvent poser problème. Mais il faut aussi s'arrêter à la première erreur que l'on croise et ne pas les empiler, ce qui peut éviter d'avoir un message d'erreur trop long, ou d'en louper une qui serait cachée dans les autres. J'ai pu effectuer un gros travail sur la gestion d'erreurs avec mes dernières missions. Elles demandaient de renvoyer une erreur si un fichier de configuration était supprimé. Et avec beaucoup d'aide car ce n'était vraiment pas une partie simple pour moi car il fallait avoir une bonne compréhension du problème et du produit, on a cherché le meilleur cas pour gérer cette erreur. Envoyer une erreur n'est donc pas forcément simple car il faut aussi qu'elle soit assez importante pour arrêter la fonction dans laquelle elle se trouve. En effet, on peut par exemple afficher un warning pour prévenir qu'il y a un problème mais qui n'est pas grave au lieu de renvoyer une erreur qui bloquerait la suite alors qu'on aurait pu faire cette suite avec la présence de cette erreur.

J'ai aussi remarqué qu'il y a des erreurs que l'on peut traiter, comme un fichier qui n'existe pas. Dans ce cas, on pourrait envoyer un message clair et concis qui pourrait même donner le nom du fichier avec un wildcard dans l'appel de la fonction du print (cf annexe 17, page

40). Mais si on ne peut pas connaître l'origine de l'erreur, alors on la renvoie sans message explicite, ce qui rend plus long le décryptage et la provenance de l'erreur.
Un point m'a particulièrement étonné, c'est que dès qu'on appelle une fonction retournant une erreur, elle est traitée au plus tôt et donne lieu à de nombreux embranchements selon la nature de l'erreur (cf annexe 18, page 41).

J'ai constaté dans mes missions que quand on modifie un bout de code, cela peut faire une cascade d'erreurs dans différents fichiers à réparer et même affecter les tests, ce qui nous fait perdre du temps. Mais cette cascade peut faire surgir une erreur qu'on avait jamais trouvé ce qui peut résoudre des problèmes importants qui auraient pu arriver plus tard.
De plus, on m'a fait comprendre que quand le code ne marche pas, il faut d'abord regarder si c'est le bon programme qu'on exécute, puis si c'est bien la bonne commande que l'on veut tester, sur le bon répertoire, et enfin vérifier le code. Cette technique évite de chercher l'erreur en premier dans le code alors qu'elle viendrait d'une commande mal écrite.

Pour finir sur les erreurs, Cyril m'a montré un aspect très important de la programmation, le Profilage des données. Cela est un terme que l'on a pas vu en cours et dont je n'avais jamais entendu parler, mais j'ai trouvé le concept très intéressant même si c'est assez compliqué à utiliser pour mon niveau. Cela est le fait, en Go, de collecter les données sur toutes les goroutines utilisées par le daemon sur un laps de temps. Ces données sont ensuite représentées sur un site Internet en localhost[27] spécialement pour le profilage (cf annexe 19, page 41). Cela permet de comprendre pourquoi le programme peut être lent et pour trouver des fuites de goroutines. Cette fuite représente les goroutines qui tournent alors qu'elles ne devraient pas, ce qui peut faire ralentir le logiciel. Cela permet ainsi de détecter des fuites de mémoire mais aussi des goroutines trop gourmandes. On peut le voir sur le graphique en fonction du temps. Le profilage a donc une place importante dans le développement d'un logiciel pour le stabiliser et l'optimiser le plus possible.

4) La structure du code et leur façon de développer

Dans mon parcours sur le développement du logiciel, j'ai quelques remarques à soulever sur le code de OpenSVC.
Tout d'abord, il est important de commenter son code pour qu'il favorise la contribution open source. J'ai remarqué qu'ils commentent seulement les fonctions publiques et celles privées qui ne sont pas triviales. En effet, si je prend comme exemple les points d'entrée d'API, chacun a son propre fichier. Il est facile de trouver ce fichier car son nom est aligné sur le nom de la fonction qu'il contient (cf annexe 20, page 42). Seulement, dans les fichiers très volumineux et avec beaucoup de fonctionnalités, il est difficile de s'y retrouver. C'est donc à ce moment-là que les commentaires apparaissent et sont très utiles pour comprendre le code. De plus, j'ai remarqué que plus une librairie est utilisée, plus les commentaires sont

[27] Quand on parle de Localhost en informatique, cela signifie que l'on cible l'ordinateur où on se trouve, du point de vue du code.

précis et travaillés. En effet, dans un code utilisé par tout le monde, chaque fonction doit être expliquée de façon claire et souvent il est accompagné d'un exemple pour illustrer le contexte (cf annexe 21, page 43). J'ai trouvé ces commentaires très pratiques quand je voulais utiliser une fonction que je ne connaissais pas.

Dans le langage C, il existe les pointeurs qui permettent de modifier la valeur réelle de l'objet et non de sa copie. J'ai plus compris ce principe quand j'ai remarqué qu'il y avait des méthodes qui prennent l'objet sans pointeur mais aussi avec dans leur code. En effet, les méthodes qui demandent à modifier l'objet prennent son pointeur, tandis que ceux qui renvoient seulement des informations ne modifient pas l'objet et n'ont donc pas besoin du pointeur.

Par la suite, j'ai dû analyser le code pour en comprendre son sens et pour pouvoir en reproduire une partie dans la mienne. J'ai pu avec cela développer mon esprit d'analyse. Mais le code que je reprends, que ce soit dans le projet ou créé par ChatGPT, j'essaye de le comprendre pour éviter de recopier un code qui n'aurait pas de sens dans le mien, surtout avec celui de chatGPT qui n'est parfois qu'un exemple.

De plus, j'ai remarqué qu'un code qui fonctionnait initialement, pouvait manifester une erreur après des changements dans d'autres sections de code. Il faut continuellement améliorer le code que l'on a fait, car il ne sera jamais parfait. Aussi, mon maître et mon tuteur travaillent sur plusieurs projets et s'occupent d'un stagiaire et d'un alternant tout en même temps, ce qui rend la charge de travail assez conséquente. Je pouvais aussi remarquer qu'ils ont une approche beaucoup plus technique quand ils parlent entre eux et cherchent des solutions assez poussées que je n'aurais jamais trouvé, mais c'est assez normal car je suis en apprentissage, je n'ai pas encore cette vision, leur expérience.

Ce qui m'a surtout étonné sur leur façon de développer, c'est que Christophe n'utilise que les terminaux pour programmer et il n'utilise même pas Terminator. Il est cependant très rapide et ce n'est pas forcément plus lent de ne travailler qu'avec des terminaux. En comparaison avec Cyril, il utilise Goland et Terminator, mais travaille avec beaucoup de fenêtres à l'écran. Cela dépend de l'habitude et de nos besoins pour travailler, en utilisant un IDE ou non. Mais je peux aussi en conclure que même dans une toute petite équipe, les différents développeurs n'utilisent pas forcément les mêmes outils pour travailler.

Pour continuer, ils utilisent peu d'outils pour s'organiser. Comme ils sont une petite équipe, ils discutent de ce qu'ils doivent faire entre eux. Ils se donnent des missions pertinentes pour le produit et qui ont de la valeur pour les utilisateurs. Il n'y a pas de rôle intermédiaire qui dirige les différentes missions qu'il faudrait faire ou qui doivent être faites. Pour organiser certaines tâches, ils utilisent Trello. C'est une application permettant d'organiser des tâches en fonction de leur avancée, que j'ai pu entrevoir dans mes cours. Ces tâches sont représentées par des cartes, et chaque carte est associée à une planche, traduisant d'un niveau d'avancement dans le projet. Ces planches contiennent par exemple des cartes en attente, à trier ou finies.

Dans le cas d'une entreprise avec beaucoup de développeurs, il est important que chaque personne ait une partie définie propre à eux et qui "match" avec leur compétence préliminaire au bon développement du produit. Contrairement aux développeurs de OpenSVC, qui ont des compétences un peu partout. Sur GitHub, il existe un outil pour gérer un projet en groupe, ce qui permettrait, si l'équipe avait plus de développeurs, de bien s'organiser.

5) L'optimisation d'un programme

L'optimisation d'un programme est très intéressante à analyser, car beaucoup de choses peuvent faire que le logiciel est plus lent et donc il est important d'y prêter attention. En effet, au niveau du collecteur, il doit être le plus efficace possible pour le traitement de données en grande quantité, il faut donc une bonne complexité algorithmique, l'efficacité du programme entre autres. Les méthodes que l'on utilise sont de l'ordre de O(1) au minimum, des opérations simples comme une addition, et de O(n) en moyenne, comme une boucle for. Mais ce qu'on voudrait éviter dans un algorithme comme le collecteur, c'est de l'ordre de $O(n^2)$, comme un parcours sur une map contenant des listes, ou plus. L'entreprise doit chercher les fonctions les plus justes pour avoir la meilleure optimisation possible. Dans un programme comme celui que l'on fait, chaque milliseconde est précieuse, et il faut éviter des appels de variables inutiles, des casts d'un type à un autre qui peut être évité, etc. Tous les petits détails de chaque ligne du code peuvent faire perdre du temps. De plus, l'IDE peut parfois aider à optimiser ce qu'on fait. J'ai pu remarquer que si une condition répète plusieurs fois le même bloc d'instruction, l'IDE peut proposer de factoriser.

J'ai aussi remarqué que le nom des variables, des fonctions et des fichiers devait correspondre exactement à son utilisation. En effet, on doit pouvoir savoir directement à quoi sert la variable, où elle apparaît, quel est son type, etc.

J'ai particulièrement apprécié étudier les algorithmes. J'aime comprendre comment ils sont construits et j'ai parfois proposé de factoriser des fonctions qui faisaient quasiment la même chose, ce qui permet d'avoir un binaire moins lourd après compilation. J'ai aussi remarqué sur mes analyses que si une partie de la fonction devient plus grosse que le reste, c'est que l'on peut diviser cette partie (cf annexe 22, page 43). Ce qui permet plus de lisibilité mais aussi un code plus réutilisable avec des séparations nettes sur chaque fonction que l'on appelle.

Pour continuer, il faut aussi prêter attention aux imports de packages au début du fichier. Dans le cas où on importe un fichier seulement pour une fonction alors qu'elle en a 50, cela oblige au programme de pré-charger les autres fonctions qui restent sans les utiliser, ce qui n'est pas optimal.

De plus, il faut essayer de réduire l'allocation de mémoire en évitant la création de variables utilisées une seule fois, il sera ainsi préférable de créer des variables que si elles doivent

être utilisées plusieurs fois. Cependant, si la définition de cette variable est très longue, on peut factoriser en découpant les différentes parties, cela créer un code plus lisible. J'ai aussi découvert qu'il fallait bien choisir le mode que l'on utilise pour ouvrir un fichier. En effet, si on veut simplement récupérer une information, on va utiliser le mode lecteur, et si on veut écrire quelque chose, on va utiliser le mode écriture. J'ai donc découvert des combinaisons de mode comme ouvrir un fichier en mode écriture avec le mode "TRUNC", qui efface tout dans le fichier puis écrit. Ce que je n'avais pas encore vu dans le cours sur le langage C. Cela permet d'avoir une optimisation sur la manipulation des fichiers pour éviter des actions inutiles.

J'ai aussi découvert le concept de sérialisation des données. Cela permet de convertir des structures de données natives du langage de développement en un format neutre, lisible sous forme de chaîne de caractère. A l'inverse, cette chaîne de caractères peut être rechargée dans le même langage ou un autre langage, sur la même machine ou sur une autre. Cette capacité est très utile pour les API qui utilisent très couramment la sérialisation JSON pour le format des requêtes et des réponses (cf annexe 23, page 44).

Pour finir, j'ai pu remarquer qu'il y avait des codes optimisés mais qui sont illisibles, et ceux qui ne sont pas optimisés sont lisibles. Cela peut être un peu paradoxal quand on doit faire un programme optimisé. Mais cela dépend ce qu'on veut faire, dans le cas de OpenSVC et de mon point de vue, ils privilégient souvent la lisibilité. L'optimisation amène à un fichier moins lourd à traiter lors de l'exécution du programme. Le problème quand c'est trop optimisé, c'est que c'est difficile de suivre le code car il est très compact. C'est ce que je vois d'un programme optimisé. Il faut donc donner un sens à l'optimisation pour justement ne pas perdre du temps à chercher quelque chose dans un code trop compact et qui finit par ne plus avoir de sens.

5) Les contraintes d'un développeur

Pendant mon stage, j'ai pu rencontrer beaucoup de problèmes qu'un développeur peut avoir. Le tout premier que j'ai subi a été mon PC qui n'était pas assez puissant pour supporter l'IDE Goland sur Linux. J'ai dû prendre l'habitude de travailler avec cela, mais j'ai à un moment songé à n'utiliser que les terminaux comme ce que fait Christophe, mais cela était trop long pour m'adapter à ce poste. Il y a aussi la redondance du code que j'ai beaucoup eu. En effet, pour créer les points d'entrée d'API, j'ai mis beaucoup de temps à recopier le code qui était quasiment le même à chaque fois, même si je me suis rédigé un chemin à suivre pour créer ces handler d'API. Mais cela ne m'a pas vraiment gêné car c'était les missions les plus amusantes à faire.

Tout au long du projet, il y a tout le temps des bugs qui apparaissent et qui empêchent d'avancer. Ce qui donne parfois beaucoup de problèmes à régler mais en général, si le logiciel est bien construit, il n'y en aura pas beaucoup en même temps.

De plus, il y a beaucoup de contraintes quand on connaît bien le produit et cela nous force à chercher la meilleure solution possible car les autres ne sont pas sûres. Je n'avais pas cette pensée avant car je ne connaissais encore rien du produit, mais j'avais un minimum d'idées pour donner un résultat convainquant en fonction de leur structure, et non comment j'aurais voulu faire.

Pour continuer, la lisibilité des informations dans un produit comme OpenSVC est très importante car la moindre erreur peut être grave. Le développeur doit ainsi être capable de traiter les bonnes informations mais aussi de les transmettre et de les afficher correctement. Sans données correctes, le client qui les reçoit peut faire de mauvaises manipulations et pourrait nous demander de l'aide, ce qui n'aurait pas été une perte de temps si l'information avait été structurée correctement.

Enfin, le développeur doit faire attention quand il manipule des fichiers sensibles. En effet, ceux qui sont confrontés au déploiement du logiciel chez les clients touchent aux serveurs et à toutes les données qui y circulent. Il ne faudrait pas supprimer ces données avec une mauvaise manipulation ou que l'agent déployé lui-même le fasse. Il faut donc penser à bien structurer et encadrer le logiciel pour pouvoir le contrôler.

Conclusion

Pour conclure sur mon expérience chez OpenSVC, elle a pu m'apporter beaucoup de connaissances sur le développement d'un logiciel. Avant mon stage, mon objectif a été de trouver un stage dans le développement de logiciel, ce que j'ai pu faire. Mais je n'aurai jamais pensé à toucher autant de domaines, que ça soit sur l'utilisation d'un nouveau langage, l'utilisation de Linux, la gestion des tests ainsi que l'utilisation de mes connaissances sur tous les cours de programmation. En effet, j'ai pu retrouver tout ce que j'avais appris en cours, et je suis conscient que je n'ai découvert qu'une infime partie de ce que j'aurais pu voir si j'avais fait 6 mois de stage, ce qui me donne encore plus envie de continuer dans cette voie. Les nombreuses connaissances que ce stage m'ont apportées sont très complémentaires avec mon parcours en BUT.

Cependant, je n'ai jamais été en contact avec le client, ce qui pourrait être intéressant à faire dans un futur stage. Car cela permet de recueillir des besoins et d'utiliser ses compétences au niveau du déploiement des services dans l'infrastructure du client et donc de bien comprendre le produit.

Pour continuer, tout le travail est sur l'ordinateur et n'est que de la programmation de logiciel, ce qui se différencie beaucoup des métiers dans l'administration ou dans le commerce où cela est plus centré sur la gestion clientèle. De plus, j'ai pu pratiquer des jobs étudiants en tant que secrétaire dans un garage, ou encore caissier à Carrefour, qui m'ont donné des expériences dans la relation clientèle et ce n'est pas du tout pareil que ce que j'ai fait dans mon stage.

Mon travail dans ce stage a ainsi apporté à l'entreprise des patchs[28] sur des tâches assez simples, mais qui lui serviront pour plus tard. Ce travail que l'on m'a délégué a été un bon moyen d'apprentissage pour moi car il n'était pas très compliqué, mais en même temps trop simple pour Christophe et Cyril, sauf quand j'ai fait les missions qui touchaient le cœur du daemon.

Pour finir, le travail que j'ai effectué pendant ce stage m'a beaucoup plu et cela m'a donné envie de continuer plus tard dans une entreprise qui travaille sur le développement d'un logiciel comme celui-ci.

La société OpenSVC est satisfaite du déroulement du stage, et reste ouverte à de nouveaux stages et alternances avec l'IUT, selon ses besoins et opportunités

[28] Ajout de code pour résoudre des problèmes ou pour ajouter une fonctionnalité au projet

Annexe

1 : Une API REST (Representational State Transfer) est une manière de décrire une API qui renvoie une structure de données en fonction d'un chemin. Cette API est sollicitée par l'appel d'une requête en HTTP.

---

2 : Le logiciel utilise des canaux de communications pour s'échanger des informations sur le cluster et pour avoir un "signe de vie" des noeuds :

OpenSVC dispose de 4 pilotes pour créer ces canaux< de communication:

- unicast : un événement est délivré tour à tour à tous ses partenaires en TCP/IP
- multicast : un événement est délivré à tous ses partenaires en un seul message multicast IP
- disk : un événement est écrit sur une zone dédiée d'un disque lisible par tous les partenaires
- relay : un événement est écrit dans une base clé-valeur par une requête API via https TCP/IP

Tant qu'un seul de ces canaux fonctionne, le cluster reste cohérent et opérationnel.

---

3 : Fonctionnement des commande sous Linux :

Une commande sous Linux prend plusieurs arguments qui vont orienter vers le type de réponse que l'on veut obtenir. La commande commence par l'emplacement du binaire en fonction de l'emplacement courant sur le système de fichier et avec un ensemble d'argument disponible. Il y a en plus des flags, drapeaux en français, qui permettent de choisir ou non de mettre un certain argument. Pour bien comprendre, il y a par exemple cette commande : "bin/om daemon start"; Cette commande permet de démarrer le daemon, elle est composé de "bin/om", l'emplacement du binaire, puis "daemon start", deux arguments pour désigner le daemon, puis le start pour faire l'action de démarrer le daemon. De plus, l'utilisation de flags dans une commande permet de rajouter une action comme dans la commande suivante : "ox node print config –node dev3n3". Ainsi, cette commande appelle le binaire ox et affiche le fichier de configuration d'un nœud. On va ensuite préciser le nœud "dev3n3" en utilisant le flag "--node". Ainsi, avec ce flag on peut par exemple donner comme arguments une liste de nœuds pour afficher le fichier de configuration de chaque nœud, comme avec cette commande "ox node print config --node 'dev3n1 dev3n2 dev3n3'".

Exemple avec "ox node print config –node dev3n3" :

```
alexandre@alexandre-Vivobook:~$ ox node print config --node dev3n3
[DEFAULT]
id = XXX

[asset]

[checks]

[compliance]

[dequeue_actions]

[disks]

[packages]

[patches]

[section]

[sysreport]
```

---

4 : Une structure d'un manifest de OpenAPI est représenté comme ceci :

```yaml
/node/name/{nodename}/system/group:
  get:
    description: View the group
    operationId: GetNodeSystemGroup
    parameters:
      - $ref: '#/components/parameters/inPathNodeName'
    responses:
      200:
        description: OK
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/GroupList'
      400:
        $ref: '#/components/responses/400'
      401:
        $ref: '#/components/responses/401'
      403:
        $ref: '#/components/responses/403'
      500:
        $ref: '#/components/responses/500'
```

Le chemin de l'API est représenté par la première ligne et tout ce qui suit sont des informations sur ce que va contenir la structure, comme les nombres 200 ou 400 qui correspondent au statut de la réponse que peut envoyer cette API. De plus, quand la

réponse à un statut 200, elle contient un objet qui peut lui-même contenir d'autres objets :

```yaml
GroupList:
  type: object
  required:
    - kind
    - items
  properties:
    kind:
      type: string
      enum:
        - GroupList
    items:
      $ref: '#/components/schemas/GroupItems'
```

Pour ma première mission et mon premier point d'entrée d'API, j'ai transformé les données venant du cache, représentées par "data.GIDS", par l'objet "GroupItems" de l'API. Ainsi, j'ai aussi transformé les objets dans les objets, comme "GroupItem" ou "Group" :

```go
items := make(api.GroupItems, len(data.GIDS))
for i := 0; i < len(data.GIDS); i++ {
    items[i] = api.GroupItem{
        Kind: "GroupItem",
        Data: api.Group{
            ID:   data.GIDS[i].ID,
            Name: data.GIDS[i].Name,
        },
        Meta: api.NodeMeta{
            Node: a.localhost,
        },
```

---

5 : Exemple de l'utilisation de la commande curl, on va passer l'URL pour accéder à un chemin d'API pour qu'il nous renvoie de la data sous forme de JSON :

```
alexandre@alexandre-Vivobook:~$ curl -o- -k -X GET -u $password https://dev3n3.opensvc.com:1215/node/name/dev3n3/system/san/initiator -s | jq
{
  "items": [
    {
      "data": {
        "name": "yes",
        "type": "iscsi"
      },
      "kind": "SANPathInitiatorItem",
      "meta": {
        "node": "dev3n3"
      }
    }
  ],
  "kind": "SANPathInitiatorList"
}
```

---

6 : Pour donner un exemple sur une commande créé à partir de cobra, j'ai pris comme exemple les commandes que j'ai créé :

```
alexandre@alexandre-Vivobook:~$ ox node system
node system commands

Usage:
  ox node system [command]

Available Commands:
  disk        show node system disks
  group       show node system groups
  hardware    show node system hardware
  ipaddress   show node system IP address
  package     show node system package
  patch       show node system patch
  property    show node system property
  san         node system san commands
  user        show node system users

Flags:
  -h, --help   help for system

Global Flags:
      --color string    Output colorization yes|no|auto. (default "auto")
      --config string   Config file (default "$HOME/.opensvc.yaml").
      --server string   URI of the opensvc api server. scheme https|tls.

Use "ox node system [command] --help" for more information about a command.
```

L'affichage ci-dessus représente un message qui liste toutes les possibilités d'arguments après "system", dans la section "Available Commands". Il y a deux colonnes, celle de gauche représente l'argument et à droite la description de la fonction de la commande avec cet argument. On peut encore descendre dans l'arbre des commandes avec l'argument "san" qui contient encore deux arguments possibles après "san", comme "ox node system san initiator".

---

7 : L'exemple ci-dessous représente un code utilisant les goroutines :

```go
for _, nodename := range nodenames {
    go func(nodename string) {
        defer func() { doneC <- nodename }()
        response, err := c.GetNodeSystemGroupWithResponse(ctx, nodename)
        if err != nil {
            errC <- err
            return
        }
        switch {
        case response.JSON200 != nil: q <- response.JSON200.Items
        case response.JSON400 != nil: errC <- fmt.Errorf( format: "%s: %s", nodename, *response.JSON400)
        case response.JSON401 != nil: errC <- fmt.Errorf( format: "%s: %s", nodename, *response.JSON401)
        case response.JSON403 != nil: errC <- fmt.Errorf( format: "%s: %s", nodename, *response.JSON403)
        case response.JSON500 != nil: errC <- fmt.Errorf( format: "%s: %s", nodename, *response.JSON500)
        default: errC <- fmt.Errorf( format: "%s: unexpected response: %s", nodename, response.Status())
        }
    }(nodename)
}
```

Dans cet exemple, on effectue une boucle sur les nœuds du cluster. La deuxième ligne représente le démarrage d'une goroutine, après cela tout ce qui est dans la fonction (représenté par "func") est exécuté par cette goroutine. Ainsi, chaque tour de boucle appelle le handler "GetNodeSystemGroupWithResponse" du nodename à l'intérieur d'une goroutine. Ce code permet de pouvoir appeler chaque handler des nœuds indépendamment les un des autres. Dans le cas où le premier nœud de la liste renvoie une erreur, il ne bloquera pas la les prochains tour de boucles.

Extrait d'un code utilisant des channels avec les "<-". Cela permet à la goroutine de faire passer un message attrapé par le "select" du for, ainsi pour chaque "case" il y aura une goroutine associé :

```go
for {
    select {
    case err := <-errC:
        errs = errors.Join(errs, err)
    case items := <-q:
        l = append(l, items...)
    case <-doneC:
        done++
        if done == todo {
            goto out
        }
    case <-ctx.Done():
        errs = errors.Join(errs, ctx.Err())
        goto out
    }
```

Exemple d'un retour de commande avec goroutines, dans le flag "--node" on a précisé dev3n2 puis dev3n3 :

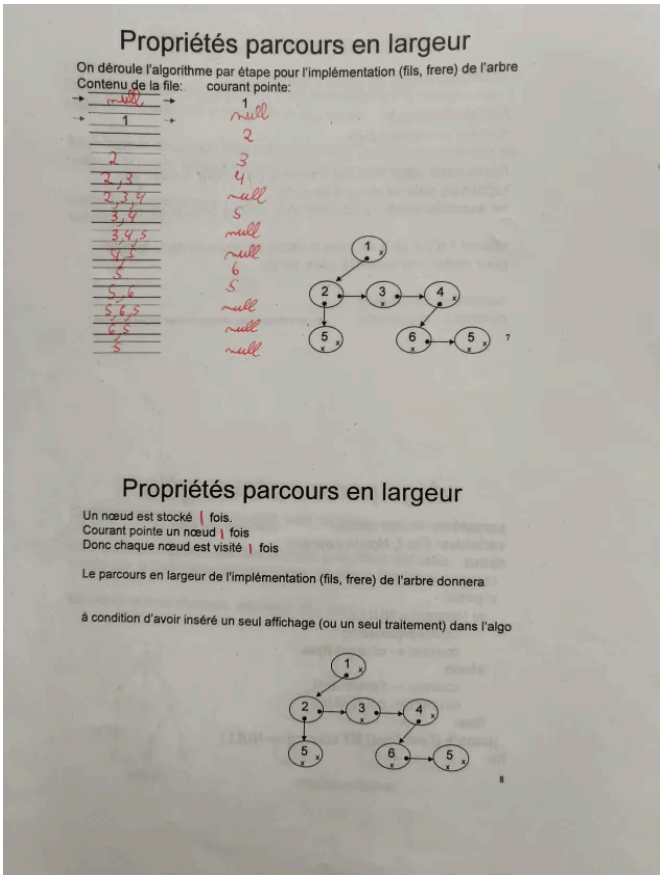
```
alexandre@alexandre-Vivobook:~$ ox node print config --node "dev3n2 dev3n3"
#
# nodename: dev3n3
#
################################################################################
[DEFAULT]
id = XXX

[asset]

[checks]

[compliance]

[dequeue_actions]

[disks]

[packages]

[patches]

[section]

[sysreport]

Error: dev3n2: unexpected response: 404 Not Found
```

Exemple d’un retour de commande sans goroutine, dans ce cas comme la boucle sur nodenames commence par dev3n2, et qu’il renvoie une erreur, on ne voit pas le résultat de dev3n3, car la boucle s’est arrêté sur dev3n2 qui a renvoyé une erreur :

```
alexandre@alexandre-Vivobook:~/GolandProjects/om3$ ox node print config --node "dev3n2 dev3n3"
Error: dev3n2: unexpected response: 404 Not Found
```

---

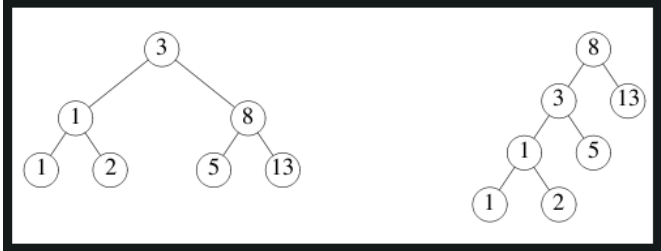
8 : Extrait de mon cours algorithme avancé représentant l’arbre que je devais traiter en Go, sous une forme que l’on a vu en cours.



Extrait de l'exercice du tutoriel :

Exercise: Equivalent Binary Trees

There can be many different binary trees with the same sequence of values stored in it. For example, here are two binary trees storing the sequence 1, 1, 2, 3, 5, 8, 13.



A function to check whether two binary trees store the same sequence is quite complex in most languages. We'll use Go's concurrency and channels to write a simple solution.

This example uses the `tree` package, which defines the type:

```
type Tree struct {
    Left  *Tree
    Value int
    Right *Tree
}
```

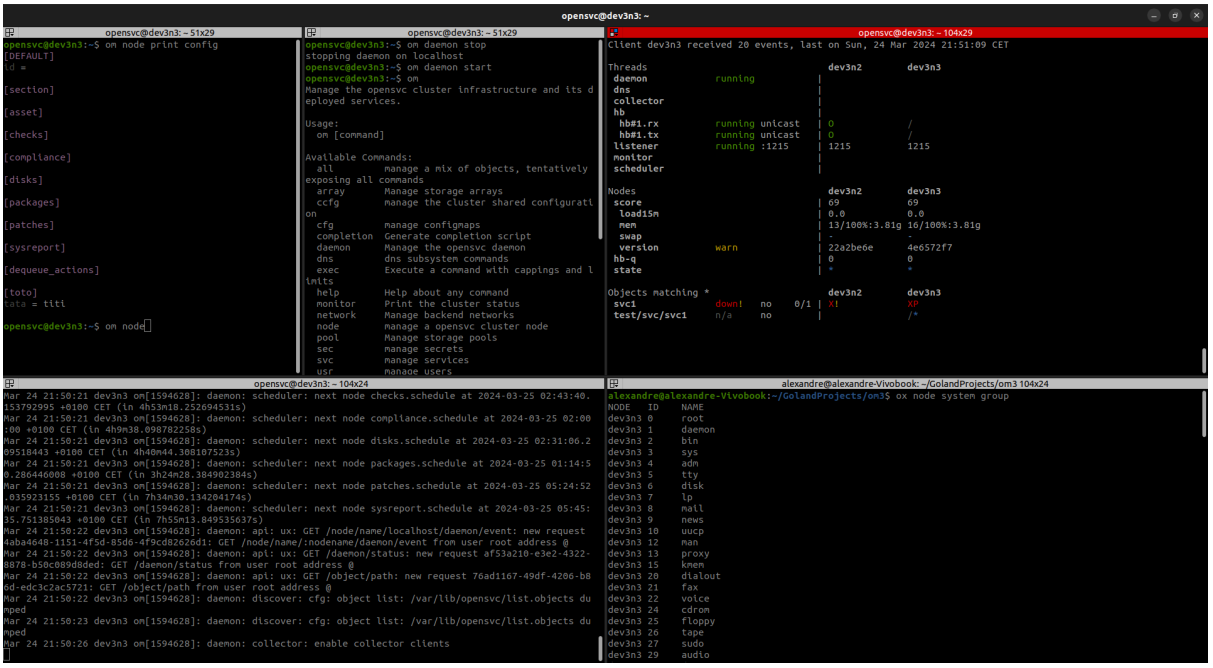
Continue description on next page.

9 : Les wildcard sons les “%s” en orange sur la ligne ci-dessous, le premier wildcard sera représenté par la variable “nodename” et le deuxième par “response.Status()”. Cela permet d’implanter dans une chaine de caractère comme “: unexpected response :” des variables qui peuvent donner d’autres types que le string (chaîne de caractère).

```
fmt.Errorf( format: "%s: unexpected response: %s", nodename, response.Status())
```

---

10 : Dans ce screenshot de mon Terminator, on peut voir que j’ai 5 terminaux ouverts, donc cela peut vite devenir incompréhensible.



---

11 : Exemple d’une déclaration de variable de type map :

```
NodesInfo map[string]NodeInfo
```

La map contient des clés de type string et les valeurs de chaque clé sont de type “NodeInfo”, un type complexe.

---

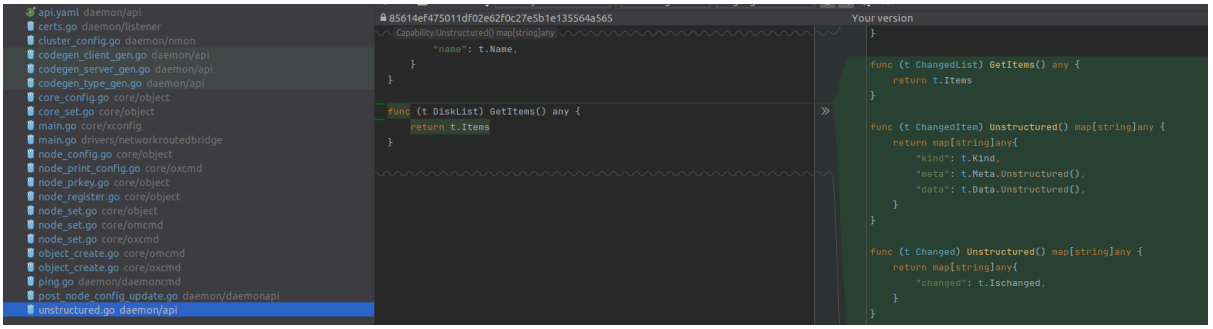
12 : Exemple d'un type struct :

```
NodeInfo struct {  3 usages  Christophe V
    Env     string     `json:"env"`
    Labels  Labels     `json:"labels"`
    Paths   san.Paths  `json:"paths"`
    Lsnr    Lsnr       `json:"lsnr"`
}
```

Le type NodeInfo est une struct contenant diverses variables avec chacune un type.

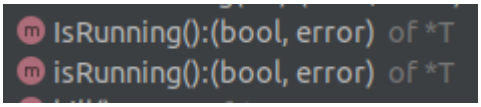
---

13 : Sur ce screenshot pris sur Goland, à gauche tous les fichiers modifiés, et dans les 2 panneaux de droite la différence entre l'ancienne version et la nouvelle version d'un fichier modifié.

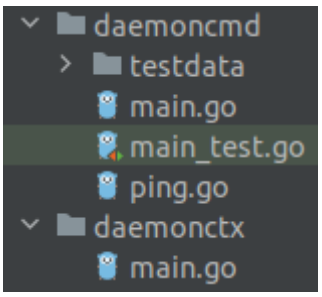


---

14 : Pour différencier le privée du public, il y a ci-dessous une fonction avec une majuscule (la première) et une en minuscule (la deuxième).



Ces fonctions sont incluses dans le package "daemoncmd" mais isRunning(), la fonction privée, ne peut pas être appelée dans le package "daemonctx".



---

15 : Pour représenter Open de la méthode SOLID, on peut prendre cette exemple :

```
func (t *T) Stop() error {  Cyril Galibern
    release, err := getLock( desc: "Stop")
    if err != nil {
        return err
    }
    defer release()
    return t.stop()
}
```

Cette fonction publique crée une sécurité pour bloquer la personne qui voudrait stopper en même temps le daemon. Mais on peut modifier la fonction publique (Open) sans modifier la fonction privée (Close).

---

16 : Dans le code ci-dessous, il y a le programme principal, puis la troisième condition “if strings.Contains(string(b))”, si elle est vraie, on renvoie une erreur de test. Mais si il n’y avait pas eu de test créé sur cette partie, il n’y aurait pas eu cette condition. Elle ne sert donc que pour les tests.

```
func isCmdlineMatchingDaemon(pid int) (bool, error) {  1 usage  Alexandre Meunier
    log := logger( s: "test:")
    log.Debugf( format: "test cmdline")
    b, err := os.ReadFile(fmt.Sprintf( format: "/proc/%d/cmdline", pid))

    if errors.Is(err, os.ErrNotExist) {
        return false, nil
    }

    if err != nil {
        return false, err
    }

    if strings.Contains(string(b),  substr: "/daemoncmd.test") {
        return true, errGoTest
    }
```

---

17 : L’erreur retournée contient un fichier qui permet de donner le contexte

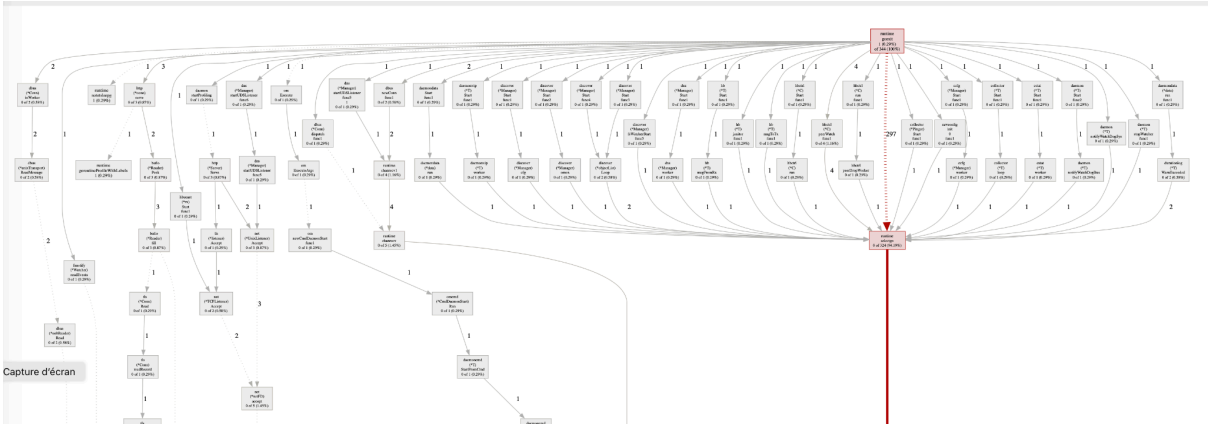
```
if errors.Is(err, os.ErrNotExist) {
    return false, fmt.Errorf( format: "%s not exist", file)
}
```

---

18 : Voir l’image de la structure du code du 16 de l’annexe (Page 40)

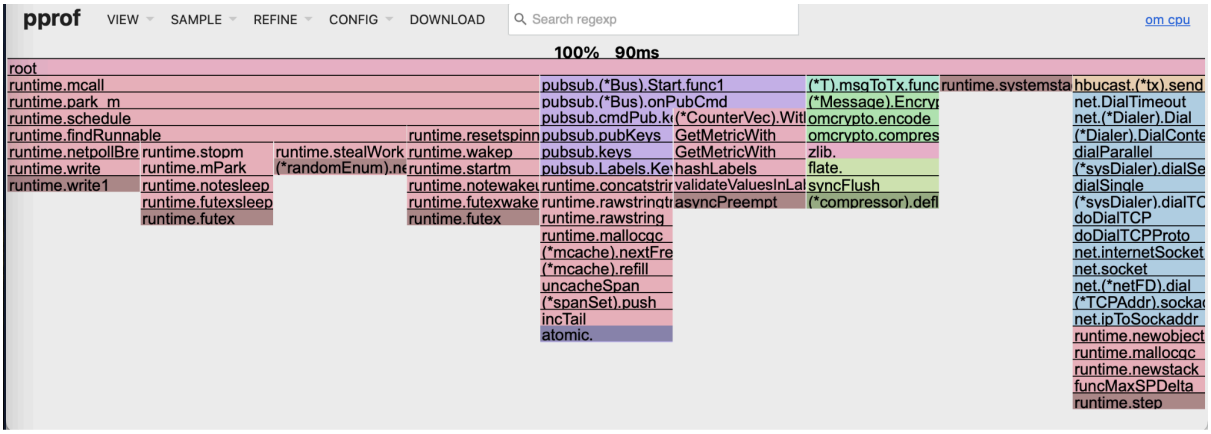
---

19 : En utilisant le module Pprof de Golang représenté par un site, on peut représenter les données des goroutines sous la forme d’arbre :



Dans cette représentation, chaque branche représente un groupe de goroutines et chaque nœud de cet arbre représente une fonction. Ainsi, on peut voir le chemin des goroutines.

On peut aussi avoir la représentation ci-dessous, l'échelle est un laps de temps, ici 90 ms, et la première ligne représente la goroutine du root, de la racine du dameon. À chaque ligne, il y a différentes goroutines créées par celles du dessus, et plus on va à droite plus on avance dans le temps. Cela permet de connaître toutes les goroutines créées dans un laps de temps.



Pour finir, on peut avoir une représentation sans graphique ci-dessous, avec sur chaque ligne, différentes données de goroutines, dont le nombre de goroutines créé par la goroutines (Flat) ou tout simplement son nom.

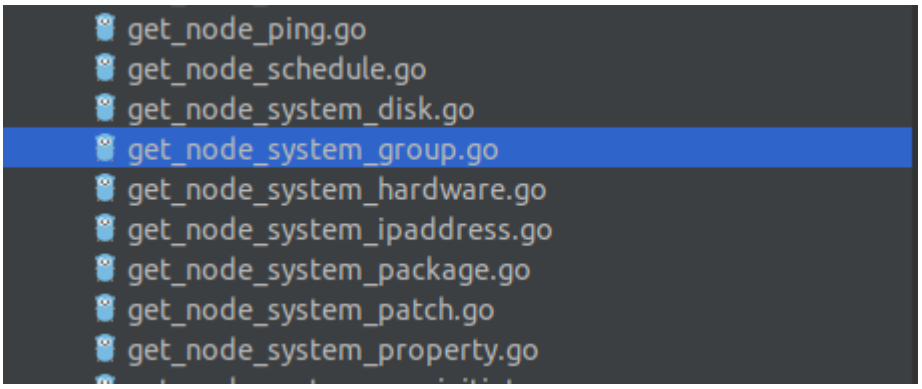
pprof VIEW REFINE CONFIG DOWNLOAD Search regexp unknown goroutine

Flat	Flat%	Sum%	Cum	Cum%	Name	Inlined?
341	99.13%	99.13%	341	99.13%	runtime.gopark	
1	0.29%	99.42%	1	0.29%	runtime.notetsleepg	
1	0.29%	99.71%	1	0.29%	runtime.goroutineProfileWithLabels	
1	0.29%	100.00%	344	100.00%	runtime.goexit	
0	0.00%	100.00%	1	0.29%	sync.runtime_Semacquire	
0	0.00%	100.00%	1	0.29%	sync.(*WaitGroup).Wait	
0	0.00%	100.00%	1	0.29%	runtime/pprof.writeRuntimeProfile	
0	0.00%	100.00%	1	0.29%	runtime/pprof.writeGoroutine	
0	0.00%	100.00%	1	0.29%	runtime/pprof.runtime_goroutineProfileWithLabels	
0	0.00%	100.00%	1	0.29%	runtime/pprof.(*Profile).WriteTo	
0	0.00%	100.00%	1	0.29%	runtime.semacquire1	

Toutes ces représentations paraissent complexes à première vue, mais cela est très utile pour analyser la consommation des goroutines du daemon.

---

20 : Dans le package des points d'entrée, chaque fichier commence par un verbe[29], puis les arguments de la sous-commande ox ou om, ce qui rend explicite la fonctionnalité de ces points d'entrée.



L'image ci-dessous représente la fonction du point d'entrée de "get_node_system_group.go" qui porte le même nom que le fichier, ce qui simplifie la compréhension de la structure du code dans cette partie.

---

[29] Un verbe en informatique désigne une méthode utilisée en HTTP. Il existe le verbe "GET", il contient des informations qui ne sont pas sensibles. On peut en voir dans les URL HTTP après le "?" comme "https://exemple/?verbe=get&information=ananas". Et le "POST" est utilisé pour des informations sensibles comme un mot de passe, un nom d'utilisateur. J'ai pu les utiliser dans les projets demandant de créer un site pendant mon BUT.

```go
func (a *DaemonAPI) GetNodeSystemGroup(ctx echo.Context, nodename api.InPathNodeName) error {
    if a.localhost == nodename {
        return a.getLocalNodeSystemGroup(ctx)
    }
    return a.proxy(ctx, nodename, func(c *client.T) (*http.Response, error) {
        return c.GetNodeSystemGroup(ctx.Request().Context(), nodename)
    })
}
```

---

21 : Exemple d'un commentaire avec exemple de la fonction "Sprintf" d'un package de base en Go.

```
Sprintf formats according to a format specifier and returns
the resulting string.

Example
  const name, age = "Kim", 22
  s := fmt.Sprintf("%s is %d years old.\n", name, age)

  io.WriteString(os.Stdout, s) // Ignoring error for simplicity.

  // Output:
  // Kim is 22 years old.
```

---

22 : Exemple d'une fonction avec un code propre. Les fonctions sont "daemonPidFile", "extractPidFromPidFile" et "isCmdLineMatchingDaemon" qui permettent une meilleure lisibilité et ainsi de factoriser le code si on veut appeler ces fonctions à l'extérieur de getPid.

```go
func (t *T) getPid() (int, error) {  2 usages
    pidFile := daemonPidFile()
    pid, err := extractPidFromPidFile(pidFile)
    if errors.Is(err, os.ErrNotExist) {
        return -1, nil
    }
    if err != nil {
        return -1, err
    }
    v, err := isCmdlineMatchingDaemon(pid)
    if !v {
        return -1, err
    }
    return pid, err
}
```

---

23 : Pour donner un exemple de sérialisation, j'ai pris le type de structure "Group" dans l'image ci-dessous (ligne 8-11)créé automatiquement à partir du manifest d'OpenAPI (Voir le 4 de l'annexe). Le Group contient un domaine "ID" de type entier (int) et un domaine "Name" de type chaîne de caractère (string). Ce qu'il y a après le type du domaine "ID" (`json:"id"`) est appelé "tag" et représente le nom que JSON doit prendre s' il y a une sérialisation sur la structure Group.

Dans la fonction main, je déclare la variable "g1" (ligne 14) qui contient une structure Group avec une valeur pour chaque domaine ("toto" et 1), puis je sérialise la variable g1 (ligne 15), et enfin je l'affiche (ligne 16).

```
 1 package main
 2
 3 import (
 4         "encoding/json"
 5         "fmt"
 6 )
 7
 8 type Group struct {
 9         ID   int `json:"id"`
10         Name string
11 }
12
13 func main() {
14         g1 := Group{Name: "toto", ID: 1}
15         b, _ := json.Marshal(g1)
16         fmt.Println(string(b))
17 }
```

Résultat du "fmt.Println(string(b))" :

{"id":1,"Name":"toto"}

Ce résultat est sous format JSON et a bien pris en compte le tag du domaine "ID" en écrivant en minuscule mais comme le domaine "Name" n'a pas de tag, la sérialisation a repris le nom du domaine, avec la majuscule.
Ce résultat peut ensuite être utilisé dans un autre langage avec une désérialisation.