

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation : Technologies de l'information et de la communication
(TIC)

Dynamic Infrastructure Support for Kollaps

Fait par

Romain Agostinelli

Sous la direction de
Prof. Marcelo, Pasin
À l'école HE-Arc

Expert externe Dr. Rafael Pires

Fribourg, HES-SO//Master, 2023

Contents

1	Introduction	1
2	Contexte	1
3	Analyse	2
3.1	Choix du langage	2
3.2	Linux Traffic Control	3
3.3	Récupération des données réseau	4
3.4	Nouvelles contraintes	5
3.5	Support du réseau sous-jacent	6
3.6	Création d'un cluster	7
3.6.1	Constitution du cluster	7
3.6.2	Système de partage d'état	8
3.6.3	Stratégie d'adhésion	8
3.7	Tests de performance réseau	9
3.7.1	Outils	9
3.7.2	Optimisation des tests	9
3.8	Répartition des émulations	16
3.8.1	Technique choisie	17
3.8.2	Attribution conservatrice	18
3.8.3	Tolérance aux pannes	21
3.9	Types d'expérimentations supportés et orchestrateur	22
3.10	Synchronisation entre parties d'expérimentation	23
3.11	Traffic Control AL et adressage	27
4	Conception	30
4.1	Infrastructure	30
4.2	Fonctionnement général et communication	32
4.3	NetHelper - Librairie d'aide aux communications réseau	33
4.4	Fonctionnement événementiel	35
4.5	CManager - Gestion du cluster	36
4.5.1	Événements du CManager	37
4.5.2	Adhésion	39
4.5.3	Détection de noeuds en échec	40
4.6	OManager - Gestion de l'orchestration	41
4.6.1	Événements de l'OManager	42
4.6.2	Rôles	43
4.7	EManager - Gestion des émulations	44
4.7.1	Communication et événements de l'EManager	45
4.8	EmulCore - Gestion locale d'une émulation	45
4.8.1	Communication avec un Reporter	46
4.8.2	Exécution et événements	47
4.9	Reporter et Monitor - Récupération des données réseau	47
4.9.1	Détection des changements d'utilisation de bande passante	48
4.10	DockHelper - Librairie d'aide aux communications avec le Docker Engine	49
4.11	Configuration	49
5	Implémentation	50
5.1	Encodage et décodage des événements par le NetHelper	50
5.2	Gestion des Timeouts dans le CManager	51
5.3	PerfCtrl et le lancement d'iPerf3	51
5.4	Ne pas tester deux fois la même combinaison	52
5.5	Épinglage de l'EmulCore par l'EManager	54

5.6	EmulCore nettoie en toutes circonstances	55
5.7	Accès parallèles aux statistiques dans le Reporter	55
5.8	Améliorations du Monitor	55
5.9	Général	57
5.9.1	Utilisation de Tokio et asynchrone	57
5.9.2	Instanciation des composants, le Runner	57
5.9.3	Rust et safe code	57
5.9.4	Division par paquets indépendants	58
5.9.5	SymMatrix	59
6	Tests	61
6.1	Environnement de test	61
6.2	Gestion du cluster	62
6.2.1	Adhésion	62
6.2.2	Retrait et redémarrage	68
6.3	Déploiement et exécution	70
7	Limitations et améliorations	78
8	Conclusion	80
8.1	Résultats et atteinte des objectifs	80
8.2	Conclusion personnelle	80
9	Déclaration d'honneur	81
10	Ressources	82
11	Annexes	85
11.1	Cahier des charges	85
11.2	Analyse	92
11.2.1	Ébauche de protocole pour une librairie de test de performance	92
11.3	Conception	92
11.3.1	Agrandissement, système événementiel	93
11.3.2	Agrandissement, boucle principale de l'OManager	94
11.4	Diagramme de classe des structures	95

1 Introduction

Ce rapport de travail de Master présente les recherches et le développement d'une plate-forme d'orchestration permettant d'offrir le support d'une infrastructure dynamique pour Kollaps.

Il parcourt les idées et les concepts mis en place ainsi que les choix faits durant le projet sans se plonger dans les détails du code.

Tout d'abord, le document expose le contexte du projet (ch. 2) qui décrit ce qu'est Kollaps et apporte des précisions sur l'objectif du projet.

Ensuite, viennent les chapitres concernant le système développé. Ils commencent par l'analyse (ch. 3) qui présente d'abord les aspects communs à la version actuelle de Kollaps et le nouveau système d'orchestration, puis se focalise sur les recherches et les solutions trouvées aux problèmes apparus durant le projet. L'analyse est suivie de la conception (ch. 4) qui décrit les fondamentaux et le principe de fonctionnement de la nouvelle plateforme, puis l'implémentation (ch. 5) qui décrit des points importants de son développement. Après l'implémentation, viennent les tests (ch. 6) qui montrent l'état de fonctionnement du nouveau système.

Une fois la documentation du système terminée, le rapport aborde les limitations et améliorations (ch. 7) possibles du nouveau Kollaps.

Finalement, une conclusion fait le bilan du projet et se termine par une synthèse personnelle de l'auteur (ch. 8). Les ressources du projet (liens vers les répertoires git, dépendances, etc.) sont disponibles au chapitre 10.

2 Contexte

Kollaps^[7, 11], un projet de recherche conjoint entre l'Université de Neuchâtel et l'IST Lisbonne, est un banc d'essai de recherche expérimentale pour évaluer les systèmes distribués à grande échelle.

Il permet de déployer des systèmes, sous forme de conteneurs, de machines virtuelles ou de processus, sur des topologies de réseau émulées. L'idée fondamentale de Kollaps est d'émuler un réseau entre les processus qui forment un système complet (plus tard appelés PT pour Processus Terminaux) sans recourir à l'émulation de matériel réseau, comme des commutateurs ou des routeurs.

Pour ce faire, Kollaps calcule en temps réel la répartition de la bande passante et la gestion du réseau émulé que produirait l'utilisation instantanée du réseau entre chacun des processus. Il utilise les résultats de ses calculs pour limiter directement les capacités réseau de chaque processus.

Kollaps fonctionne sous forme d'expérimentation. Une expérimentation décrit une topologie réseau à émuler, comprenant les processus du système distribué (processus terminaux), des éléments réseau (ponts, routeurs) ainsi que les liens entre les éléments du réseau. Une expérimentation comporte aussi un système d'événements qui permet de définir certaines actions à exécuter à un délai défini, comme l'entrée d'un processus et sa sortie de l'expérimentation.

Kollaps est distribué et décentralisé. Pour chaque processus terminal, il y a un EmulationCore (composant du Kollaps actuel) qui se charge de calculer l'utilisation réseau du processus auquel il est affilié, ainsi que de modifier ses capacités réseau en fonction des calculs. L'EmulationCore échange des informations avec les autres EmulationCore afin de se coordonner.

Actuellement, Kollaps n'est pas sous forme de service. Il n'y a pas de serveur Kollaps auquel il faut soumettre une expérimentation. Le principe est de créer une topologie, puis démarrer Kollaps. Kollaps lit la topologie et prépare des fichiers de configuration en fonction de l'orchestrateur sur lequel la topologie est déployée (Kubernetes ou Swarm). Dans ces fichiers de déploiement, nous retrouvons les processus terminaux sous forme de conteneurs. Dans ces conteneurs sont aussi injectés les EmulationCores. Un conteneur contient alors un PT et son EmulationCore. Une fois les fichiers de configuration générés, ils sont donnés à l'orchestrateur. Une expérimentation vit alors par elle-même sur l'orchestrateur.

Le cluster de machines sur lequel repose un orchestrateur est actuellement statiquement défini et ne permet pas de changements dynamiques. Cependant, de tels changements peuvent survenir soudainement, par exemple en raison de défaillances matérielles ou logicielles.

L'objectif du projet est de transformer Kollaps en service distribué et décentralisé sur un cluster de machines, et permettre au service de supporter des changements dynamiques du cluster. Des capacités de calcul supplémentaires peuvent également être fournies par des nœuds proposés dans le cadre d'un modèle de calcul volontaire (par exemple, par des clients tiers qui ne rejoignent le réseau Kollaps que pour une durée limitée).

Pour répondre à ces objectifs, une nouvelle version de Kollaps est développée, intégrant les capacités de former un cluster et d'orchestrer des expérimentations. Il s'agit d'une nouvelle plateforme d'orchestration dédiée et complètement intégrée à Kollaps.

Le chapitre 3.4 revient sur les nouvelles contraintes qu'apporte le projet et décrit plus en détails ce qu'implique la mise en place d'une telle plateforme. Mais d'abord, les premiers chapitres de l'analyse décrivent des technologies déjà utilisées par Kollaps et qui lui permettent d'atteindre ses objectifs.

3 Analyse

L'analyse englobe les recherches sur différentes technologies utilisées dans ce projet et les recherches fondamentales sur certains problèmes qui sont apparus durant la conception du système.

La première partie décrit les technologies utilisées dans la version de Kollaps actuelle et la nouvelle version produite au long de ce projet. Ensuite, le chapitre 3.4 détaille les différences entre le Kollaps actuel et le Kollaps de ce projet. Les chapitres suivants se focalisent alors sur les contraintes liées à ce projet en particulier.

Les termes *Kollaps actuel*, *ancienne version de Kollaps* ou *version originale*, font référence à la version publique disponible de Kollaps et développée avant le début de ce projet. Le terme *nouvelle version de Kollaps* décrit le logiciel écrit durant ce projet, qui, diffère beaucoup de la version originale.

3.1 Choix du langage

Le langage de programmation utilisé pour ce projet est [Rust](#)¹. C'est un langage de type *General Purpose* qui est destiné à tous les domaines. Il a une particularité : son "Borrow Checker". Il permet de vérifier lors de la compilation si une variable n'a pas plus d'une référence mutable et immuable simultanément. Il fonctionne avec un système d'appartenance de variable. C'est un langage qui demande du temps pour le maîtriser et qui change les codes conventionnels de la programmation.

¹<https://www.rust-lang.org/>

Ses principales forces sont la performance (comparable à C) et sa fiabilité due au Borrow Checker.

Le projet Kollaps a été migré du langage Python au langage Rust récemment (2022) pour ses qualités. L'implémentation actuelle de Kollaps, due à sa transition récente, ne suis pas encore toutes les bonnes pratiques du langage et demanderait du *refactoring*.

Le but est de continuer l'implémentation avec ce langage afin de bénéficier notamment de la performance.

Le corps de Kollaps est écrit en Rust, mais il dépend encore de certaines librairies plus anciennes qui n'ont pas encore été migrées. Ces librairies permettent justement de discuter avec le noyau Linux et de modifier le filtrage des paquets réseau grâce au linux traffic control (TC).

3.2 Linux Traffic Control

Le linux traffic control est le nom donné au système de queues et aux mécanismes utilisés lorsqu'un paquet réseau doit entrer ou sortir d'une machine sous Linux. Il offre les possibilités de modifier les propriétés réseau en fonction d'un système de classification des paquets. Il permet aussi de changer le type de queue utilisé pour certains paquets et de modifier les priorités données à chacun des paquets.

Il est très utile lorsqu'il s'agit de faire du QoS pour des grands réseaux. Par exemple, le logiciel [OpenVSwitch](#)², qui permet de simuler un commutateur, utilise le TC pour effectuer du QoS[2].

Il n'existe pas aujourd'hui de très bonne documentation sur le fonctionnement et l'utilisation du TC. Après quelques recherches, deux références sortent du lot. Il s'agit de deux documentations, une parlant du TC de manière plus générale : [Traffic Control HOWTO](#)³ et l'autre de manière plus poussée : [Linux Advanced Routing & Traffic Control HOWTO](#)⁴.

Sous Linux, il y a la possibilité de créer une abstraction d'une ressource globale du système qui permet de la présenter à un ou plusieurs processus comme étant une instance indépendante de la ressource globale[13]. Il s'agit des espaces de noms, ou *namespaces* dans la langue originale. Il est donc possible de présenter le système de contrôle du trafic à un processus comme s'il s'agissait d'une instance dédiée. Cela permet de configurer le contrôle du trafic pour chacun des namespaces existants, comme si à chaque fois, nous avions une nouvelle instance du linux traffic control.

C'est le pilier du fonctionnement de Kollaps. Il utilise le fait qu'un processus présent dans une expérimentation se trouve dans un espace de nom indépendant des autres processus sur une même machine et modifie les propriétés du linux traffic control pour cet espace de nom. De cette manière, il module le réseau pour un seul processus sans impacter les autres qui se trouvent dans d'autres namespaces.

Il est possible de faire entrer ou sortir un processus dans un certain namespace avec la commande [nsenter](#)⁵[12].

Lorsqu'un processus entre dans un espace de nom (ici, plus particulièrement le Network Namespace), il partage donc la même configuration réseau que tous les autres processus à l'intérieur de l'espace. Avec des outils d'analyse de paquets, les paquets accessibles sont aussi ceux destinés à l'espace de nom dans lequel se trouve l'analyseur.

²<https://www.openvswitch.org/>

³<https://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html>

⁴<https://tldp.org/HOWTO/Adv-Routing-HOWTO/index.html>

⁵<https://man7.org/linux/man-pages/man1/nsenter.1.html>

Nous pouvons donc isoler directement des applications et insérer dans le même espace un analyseur de paquets afin de récupérer les données réseau d'une application (ou plutôt d'un namespace) en particulier.

3.3 Récupération des données réseau

Kollaps a besoin de récupérer l'utilisation de la bande passante des applications ou processus présents dans une expérimentation. Ces données permettent ensuite d'adapter la configuration du linux traffic control pour toutes les applications de l'expérimentation en temps réel afin de s'assurer que la bande passante émulée ne soit pas dépassée, mais qu'elle soit aussi utilisée au maximum.

Pour ce faire, Kollaps utilise la technologie BPF. BPF signifie Berkeley Packet Filter. Cette technologie permet à un processus en espace utilisateur de fournir un programme de filtre de paquets. Ce programme de filtre va définir quels paquets il faut renvoyer au processus utilisateur.

Lorsqu'un processus utilisateur fournit un programme de filtre, le programme est analysé puis lancé dans l'espace noyau[4].

Cela permet de trier directement quels paquets le processus utilisateur a besoin et permet une meilleure performance des processus utilisateurs, car ils reçoivent uniquement les paquets qu'ils désirent.

L'eBPF est la version "Extended" du BPF. Il s'agit aujourd'hui de la dénomination utilisée, car le projet BPF a continué d'évoluer très rapidement pour devenir eBPF.

Les programmes eBPF sont lancés en mode bac à sable directement dans le noyau, ce qui permet d'exécuter ces programmes en sécurité avec les mêmes accès aux ressources que le noyau[4].

La Figure 1 illustre la mise en place d'un logiciel eBPF communiquant avec le logiciel principal en espace utilisateur.

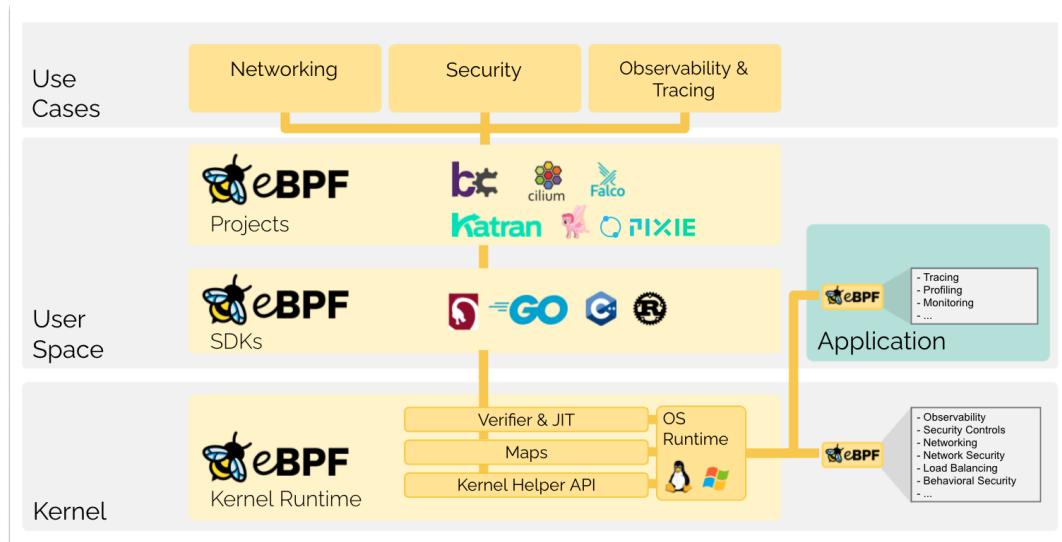


Figure 1: Structure d'une installation eBPF

Source: <https://ebpf.io/>

Après avoir fait le point sur ce qu'utilise l'implémentation actuelle de Kollaps, les chapitres

suivants se focalisent sur la modification du logiciel, ce que le projet apporte et les éléments qui en découlent.

3.4 Nouvelles contraintes

L'adaptation de l'outil Kollaps permettant le support d'une infrastructure dynamique apporte de nouvelles contraintes sur le logiciel et son environnement de fonctionnement.

Une expérimentation Kollaps peut se dérouler sur plusieurs nœuds physiques différents. Par exemple, dans un environnement Kubernetes, ces nœuds sont les *Nodes Kubernetes* qui forment le cluster.

Jusqu'à présent, l'outil considère que les connexions entre chacun des nœuds physiques utilisés dans l'environnement Kollaps ont des propriétés optimales. C'est-à-dire, une bande passante considérée comme illimitée, une latence nulle et aucune perte de données. Cela simplifie la mise en place d'une expérimentation distribuée, car il n'y a pas besoin de se soucier de l'état du réseau sous-jacent.

Un point important abordé dans ce projet est le support, non seulement d'une infrastructure dynamique, mais aussi de l'hétérogénéité de celle-ci. Cela signifie que nous voulons supporter des nœuds offrant différentes capacités, mais aussi avec des propriétés de connexion variées. Nous ne considérons donc plus le réseau sous-jacent comme un réseau avec des propriétés optimales, mais avec des capacités différentes de connexion entre chacun des nœuds.

Ce projet se rapproche d'une idée de calcul volontaire, où des nœuds physiques peuvent rejoindre et quitter un cluster pour offrir une certaine quantité de calcul et, dans notre cas, de bande passante. Cela implique que Kollaps doit dorénavant permettre de supporter l'ajout et le retrait de nœuds à la volée lors de son exécution.

Un déploiement Kollaps actuel est destiné à une seule expérimentation. Cela signifie que nous définissons une expérimentation et Kollaps se lance et se termine à la fin de celle-ci. Ce projet change la manière dont Kollaps fonctionne. Kollaps se transforme pour devenir une application de type service, et fonctionne tout le temps. Il est ensuite possible de lui fournir des expérimentations à exécuter et celui-ci se charge de les répartir sur les différents nœuds qui forment le cluster. De cette manière, Kollaps permet de supporter plusieurs expérimentations simultanées.

En résumé, voici les nouvelles contraintes et les changements qu'apporte ce projet à Kollaps :

- Support d'un réseau hétérogène entre les nœuds.
- Support d'un réseau de calcul de type volontaire avec ajout et suppression de nœuds à la volée.
- Support de plusieurs expérimentations simultanées.
- Transformation afin de devenir un service.

Ce projet n'a pas pour but de modifier le fonctionnement d'un orchestrateur existant (par exemple Kubernetes ou Docker Swarm), mais bel et bien de créer un nouveau système d'orchestration qui permet de supporter l'ajout et la suppression de nouveaux nœuds dans un cluster (comme l'ajout et la suppression de nœuds Kubernetes) et de supporter le fait que les connexions entre les nœuds du cluster aient des capacités variées.

Certains termes fréquemment utilisés dans la suite de la thèse sont décrits ci-dessous :

- Nœud Kollaps : machine physique pouvant entrer et sortir du cluster Kollaps fournant de la puissance de calcul. Analogie à un nœud Kubernetes.

- Cluster Kollaps : regroupement de nœuds Kollaps collaborant pour fournir une puissance de calcul dédiée à émuler des expérimentations Kollaps. Analogie à un cluster Kubernetes.

Les chapitres suivants se concentrent sur la création d'un tel cluster et comment supporter ces nouvelles contraintes. Ils parlent aussi des différents types de réseaux physiques et leur support pour cette nouvelle version de Kollaps.

3.5 Support du réseau sous-jacent

Afin de supporter des déploiements d'expérimentations sur un regroupement de machines, il est nécessaire d'assurer que les propriétés de connexion entre ces machines sont suffisantes pour les expérimentations. Typiquement, il n'est pas possible de déployer une expérimentation simulant une connexion de 100 Mbit/s entre deux processus terminaux répartis sur deux nœuds physiques différents n'ayant que 10 Mbit/s comme connexion physique.

Kollaps doit procéder à une découverte du réseau sous-jacent pour connaître les propriétés des liens interconnectant les nœuds du cluster. En connaissant le réseau, Kollaps peut ensuite répartir les expérimentations en s'assurant que les connexions simulées pourront atteindre leurs objectifs.

Nous distinguons deux topologies réseaux principales : les réseaux commutés, constitués uniquement de commutateurs (*switches*) et les réseaux routés, constitués de commutateurs et de routeurs.

La particularité d'un réseau routé, par rapport à un réseau commuté, est le fait qu'il existe plusieurs routes entre deux points de terminaison. Cela rend la tâche de la découverte du réseau sous-jacent très difficile.

Contrairement aux réseaux routés, les réseaux commutés offrent une seule route pour connecter deux points de terminaison. Cela est dû à la mise en place de l'algorithme Spanning Tree par les commutateurs, qui forme un arbre à partir du réseau. Un arbre implique qu'il existe un seul chemin entre chacune des machines du réseau. Cela permet de faciliter la découverte du réseau, car une fois qu'un test de performance est effectué entre deux machines, nous sommes garantis que le résultat restera le même, car les paquets suivront toujours le même chemin (dans un réseau stable).

Il est donc beaucoup plus facile de découvrir les propriétés de connexion entre deux nœuds terminaux dans un réseau commuté.

Plusieurs techniques de découverte du réseau ont été analysées.

La première consiste à effectuer des tests de performance entre chacun des nœuds. Cette technique permet de trouver les capacités réseau entre deux nœuds par l'exécution d'un test de performance sans se soucier de la topologie du réseau physique. Cela a pour avantage de ne pas devoir procéder à de la découverte de réseau. Aussi, cette technique fonctionne seulement si la route des paquets ne change pas.

La deuxième est la récupération d'informations réseau via certains protocoles utilisés par les équipements réseau pour communiquer (p. ex. SNMP) et s'échanger des informations. Typiquement, il serait aussi possible de récupérer les paquets de routage OSPF ou EIGRP (ou autres protocoles) afin de construire sa propre table de routage et découvrir le réseau sous-jacent. Cette technique a l'avantage de supporter aussi bien les réseaux routés que commutés. Cependant, il est aujourd'hui très difficile de découvrir la topologie réseau de couche 2 [1].

C'est pourquoi dans ce projet, c'est la première solution qui est choisie. Afin de découvrir les capacités de connexion entre plusieurs machines, des tests de performance seront

exécutés. Ce choix restreint aussi le réseau à être un réseau commuté, mais pas forcément à faire partie du même sous-réseau (configuration VLAN). Il est cependant possible de connecter plusieurs sites via un VPN ou autre technique de *tunneling*, typiquement via L2TPv3 [5] comme le montre le graphique ci-dessous (Figure 2).

Effectuer des tests de performance pour des petits clusters est faisable, mais lorsque le cluster grandit, chaque nœud doit être testé avec tous les autres nœuds du cluster. Cette technique implique intuitivement de faire, pour un cluster de n nœuds,

$$(n - 1) + (n - 2) + \dots + 1$$

tests, ce qui donne

$$(n - 1) \cdot \frac{1 + (n - 1)}{2}$$

tests. Cela signifie que pour un cluster constitué de 60 nœuds, il faut exécuter 1770 tests qui prennent tous potentiellement 15 à 20 secondes. C'est pourquoi une technique a été développée pour limiter le nombre de tests nécessaires. Cette technique est décrite dans le chapitre 3.7.2.

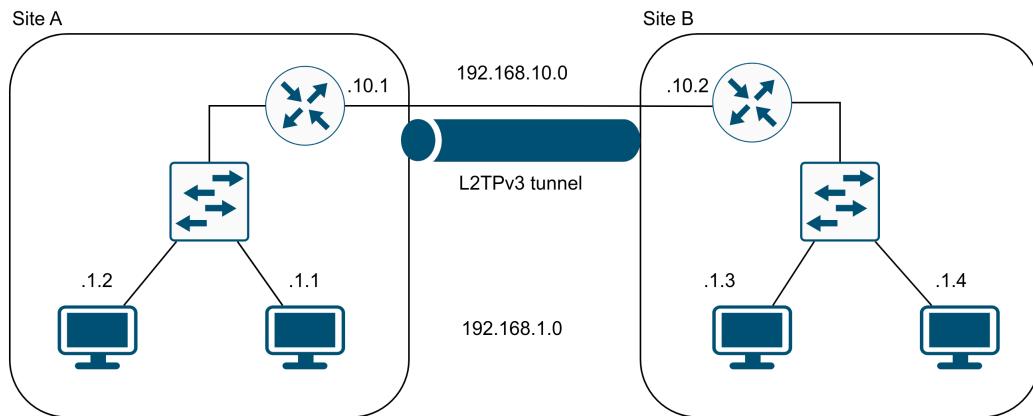


Figure 2: Réseau supporté par Kollaps partagé sur deux sites : A et B

Le chapitre suivant s'éloigne des propriétés réseau pour s'orienter sur les propriétés d'un regroupement de machines et la spécification du cluster choisie.

3.6 Création d'un cluster

La création et le maintien de l'état d'un cluster est nécessaire pour offrir une plateforme de calcul volontaire à Kollaps. Les sous-sections suivantes abordent la constitution d'un cluster, son système de partage d'état et la stratégie d'adhésion au cluster pour de nouveaux nœuds.

3.6.1 Constitution du cluster

Dans de tels regroupements de machines, où le but est de distribuer des tâches, le regroupement contient, dans la plupart des cas, un ou plusieurs *leader*. Ce *leader* a pour but de gérer le cluster et de maintenir l'état du cluster. Cela est typiquement le cas pour Zookeeper où un leader est choisi entre les nœuds[6]. Dans notre cas, la présence d'un leader permet le maintien d'un état global et de gérer l'adhésion des différents nœuds au cluster.

Il existe plusieurs algorithmes permettant d'élire un leader dans un système distribué[19].

Dans les premières expérimentations et conceptions pour ce projet, la gestion du cluster a été pensée pour supporter un changement de leader. Comme les données d'un cluster doivent être transférées entre le leader et un nouveau nœud faisant une demande d'adhésion, la sélection du leader était basée sur les meilleures propriétés réseau en moyenne entre un nœud et tous les autres. Ceci avait pour but de définir un leader qui pouvait partager plus facilement et rapidement les données du cluster à un nouveau nœud afin que celui-ci puisse effectuer la procédure d'adhésion. À chaque changement de leader, les données du cluster étaient transférées entre le nouveau nœud et le nouveau leader du cluster. De cette manière, uniquement le leader actuel avait une copie finale de l'état du cluster. L'état n'était pas partagé, mais transféré entre leaders à chaque changement.

Finalement, il a été décidé de ne pas utiliser d'algorithme d'élection de leader, mais de définir manuellement le leader du cluster. Comme cette nouvelle version de Kollaps doit aussi se charger de répartir des émulations et de maintenir leurs états, cela impliquerait d'utiliser ou d'implémenter un système de partage d'état afin de supporter un potentiel changement de leader. De plus, comme les nœuds peuvent à tout moment rejoindre ou quitter le regroupement, il devient plus complexe de supporter un changement de leader(s) dans un cas de déconnexion. Cette partie de recherche pour ce projet est mise de côté, n'étant pas l'objectif principal.

Le cluster Kollaps possède donc un seul leader qui doit être le premier nœud lancé. La solution optimale est l'utilisation d'un système d'état partagé.

3.6.2 Système de partage d'état

Comme discuté dans le chapitre précédent, il serait optimal pour le support d'algorithmes d'élection de leader d'avoir un système d'état partagé.

La mise en place d'un tel système permettrait d'améliorer la tolérance aux fautes si un leader tombe en échec, la distribution et la redondance de l'information et d'avoir un point d'accès commun à l'état du cluster. Il existe déjà des plateformes ou des outils fournissant ces fonctionnalités. Typiquement, nous pouvons citer [ZooKeeper](#)⁶ ou [etcd](#)⁷. Ces logiciels permettent de garantir un état partagé entre les différents éléments du cluster.

L'utilisation de ce type d'outil dans Kollaps permettrait une amélioration au niveau de la tolérance aux pannes et constituerait une bonne suite de projet. Ce projet n'utilise pas ces logiciels pour manque de temps.

Comme nous l'avons décrit, un nouveau nœud doit effectuer plusieurs tâches avant d'être accepté dans le cluster. Cette procédure d'adhésion est décrite dans le chapitre suivant.

3.6.3 Stratégie d'adhésion

Comme décrit dans le chapitre 3.5, cette nouvelle version de Kollaps fonctionne sur un seul sous-réseau. Cela permet notamment d'utiliser des protocoles comme UDP pour faire du multicast de sous-réseau.

Lorsqu'un nouveau nœud souhaite se joindre au cluster, il peut utiliser un multicast UDP pour demander qui est le leader actuel et s'il peut entrer dans le cluster. Uniquement le leader répond en acceptant ou en refusant la demande d'adhésion. Il se peut qu'un paquet UDP soit perdu ou erroné, c'est pourquoi un nouveau nœud doit avoir un certain nombre d'essais défini pour contrer ceci.

⁶<https://zookeeper.apache.org/>

⁷<https://etcd.io/>

Le leader peut faire le choix de rejeter une requête s'il y a déjà une autre procédure d'adhésion en cours. Effectivement, quand un nœud s'ajoute au cluster, il doit effectuer des tests de performance entre lui et les autres nœuds déjà présents dans le cluster. Si deux nœuds sont simultanément en train de s'ajouter, les résultats des tests peuvent être faussés, car la bande passante doit être partagée.

3.7 Tests de performance réseau

Les tests de performance forment un point névralgique de la constitution d'un cluster pour Kollaps. Sans ces tests, il serait impossible de garantir la précision d'une expérimentation répartie sur plusieurs machines. Kollaps doit connaître les propriétés du réseau sous-jacent.

3.7.1 Outils

Aujourd'hui, plusieurs outils sont disponibles pour effectuer des tests de performance. Dans la majorité des cas, l'outil le plus cité pour effectuer des tests réseau est [iPerf3](#)⁸. Il s'agit aussi d'un des outils utilisés par l'équipe Kollaps pour valider leur logiciel.

iPerf3 permet de tester la bande passante entre deux machines et utilise par défaut le protocole TCP pour transférer des données. iPerf3 donne comme résultat la moyenne de plusieurs transferts. Lors d'une limitation de bande passante avec des outils de Quality of Service (QoS) tel que le linux traffic control (TC), nous remarquons qu'il faut un temps de stabilisation à iPerf3 pour nous donner les bons résultats. Cela est dû à la stabilisation de la fenêtre TCP. C'est pourquoi il est possible de voir dans les tests une phase de transition lors de la limitation de la bande passante via le TC.

Pour ce projet, nous utilisons donc iPerf3 pour effectuer les tests de performance lors de la procédure d'adhésion. Les commandes nécessaires doivent être lancées par chacun des nœuds. Ceci peut être fait via un protocole défini permettant de synchroniser le lancement du serveur iPerf3 sur le nœud de destination et la partie cliente sur le nœud source du test.

Le développement d'une bibliothèque de tests de performance propre à Kollaps permettrait de mieux intégrer ces tests dans cette nouvelle version du logiciel. Aussi, elle permettrait d'effectuer des tests sur des éléments très spécifiques comme la perte de données, la latence ainsi que la gigue. Cette recherche du bon outil pour ces tests de performance a amené une ébauche d'un protocole de communication permettant la création d'une bibliothèque de tests dédiée à Kollaps. Cette ébauche de protocole est montrée dans le graphique présent en annexe au chapitre 11.2.1 (Figure 50).

Indépendamment de l'outil, un test de performance prend du temps. En général, avec iPerf3, il faut 15 secondes pour obtenir un résultat correct. Le problème du nombre de tests est introduit dans le chapitre 3.5. Le chapitre suivant traite de la manière dont ce problème est abordé et la technique développée pour limiter le nombre de tests nécessaires.

3.7.2 Optimisation des tests

En reprenant les équations décrites au chapitre 3.5, nous pouvons dire qu'intuitivement, il faudrait, pour un cluster de n nœuds,

$$(n - 1) \cdot \frac{1 + (n - 1)}{2}$$

tests.

⁸<https://iperf.fr/>

Ceci n'est pas viable. Plus le cluster grandit, plus le nombre de tests à exécuter à chaque nouvelle adhésion augmente. Il deviendrait alors impossible, à partir d'une certaine taille, à un nouveau nœud de rejoindre le cluster dû au nombre de tests qu'il doit exécuter.

Nous pouvons cependant nous baser sur certaines propriétés du réseau sous-jacent pour limiter le nombre de tests à exécuter. Une propriété utile d'un réseau commuté est que le réseau peut être représenté sous forme d'un arbre. Ce chapitre montre la technique utilisée ainsi que les résultats obtenus pour limiter le nombre de tests de performance nécessaires. La technique est la suivante :

Nous considérons trois nœuds dans un arbre : a , b et c , ainsi que trois chemins entre ces nœuds : p_1 , p_2 et p_3 avec

- a) $a \xrightarrow{p_1} b$
- b) $b \xrightarrow{p_2} c$
- c) $a \xrightarrow{p_3} c$

Le poids d'un chemin : $W(path)$ est la capacité minimale entre toutes les arêtes (edges) qui forment le chemin (la bande passante).

Comme le chemin d'un nœud à un autre dans un arbre est unique, nous pouvons considérer deux types différents d'arbres.

Le premier type est celui où p_1 et p_2 partagent au moins une arête (représenté par T_1) et le second type est celui où p_1 et p_2 ne partagent aucune arête (représenté par T_2).

Il existe plusieurs implémentations d'arbres représentant ces deux types, mais cela ne fait pas de différence pour le développement de cette explication. Nous choisissons les deux représentations suivantes pour les deux types (Figure 3) :

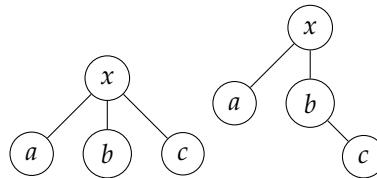


Figure 3: À gauche : un arbre implémentant T_1 . À droite : un arbre implémentant T_2

Nous pouvons décrire le chemin p_3 de cette manière pour les deux types d'arbre :

$$\begin{aligned} T_1 : a \xrightarrow{p_3} c &= (a \xrightarrow{p_1} b - (p_1 \cap p_2)) + (b \xrightarrow{p_2} c - (p_1 \cap p_2)) = a \xrightarrow{p_1} b + b \xrightarrow{p_2} c - 2 \cdot (p_1 \cap p_2) \\ T_2 : a \xrightarrow{p_3} c &= a \xrightarrow{p_1} b + b \xrightarrow{p_2} c \end{aligned}$$

Notre cas d'utilisation est de connaître la bande passante entre deux nœuds, tout en sachant le poids des deux autres chemins.

Pour le type d'arbre T_1 , il y a deux facteurs qui peuvent limiter la bande passante :

1. $a \xrightarrow{p_1} b - (p_1 \cap p_2)$
2. $b \xrightarrow{p_2} c - (p_1 \cap p_2)$

Dans le second type d'arbre (T_2), il y a aussi deux facteurs qui peuvent limiter la bande passante :

1. $a \xrightarrow{p_1} b$
2. $b \xrightarrow{p_2} c$

Nous pouvons définir le poids de p_3 comme suit :

$$T_1 : W(p_3) = \min(W(p_1 - (p_1 \cap p_2)), W(p_2 - (p_1 \cap p_2)))$$

$$T_2 : W(p_3) = \min(W(p_1), W(p_2))$$

Le but est de trouver $W(p_3)$ sachant $W(p_1)$ et $W(p_2)$. Dans ce cas, nous pouvons définir trois possibilités qui s'appliquent chacune aux deux types d'arbres.

- a) $W(p_1) = W(p_2)$

T_1 : Nous ne pouvons rien dire pour $W(p_3)$ car le facteur limitant pourrait être $W(p_1 \cap p_2)$ que nous n'utilisons pas dans p_3 .

T_2 : $W(p_3) = W(p_1) = W(p_2)$.

- b) $W(p_1) > W(p_2)$

T_1 : $W(p_3) = W(p_2)$ du fait que nous sachons que $p_1 \cap p_2$ n'est pas un chemin limitant (si c'était le cas, $W(p_1)$ serait égal à $W(p_2)$), mais le chemin limitant est $p_2 - (p_1 \cap p_2)$

T_2 : $W(p_3) = W(p_2)$ car le chemin limitant est p_2 .

- c) $W(p_1) < W(p_2)$

T_1 : $W(p_3) = W(p_1)$ du fait que nous sachons que $p_1 \cap p_2$ n'est pas un chemin limitant (si c'était le cas, $W(p_1)$ serait égal à $W(p_2)$), mais le chemin limitant est $p_1 - (p_1 \cap p_2)$

T_2 : $W(p_3) = W(p_1)$ car le chemin limitant est p_1 .

Cela signifie que si notre but est de trouver la bande passante (le poids du chemin) entre les noeuds a et c , nous pouvons chercher un noeud b avec une bande passante connue entre $a \sim b$ et $b \sim c$. Si ces bandes passantes sont les mêmes et que nous sommes dans une configuration d'arbre de type T_2 , ou que les bandes passantes sont différentes, nous pouvons définir la bande passante $a \sim c$ sans avoir à la tester.

En pratique, nous pouvons utiliser ces propriétés pour définir un algorithme qui nous dit quel test de performance doit être exécuté afin de compléter (pas forcément finaliser) un graphe montrant la bande passante entre chacun des noeuds terminaux d'un réseau.

Dans notre cas, nous voulons qu'un nouveau noeud s'ajoutant au cluster Kollaps puisse interroger cet algorithme afin de connaître les destinations (autres noeuds déjà dans le cluster) des tests de bande passante qu'il doit exécuter. Le but est de compléter le graphe des bandes passantes afin de connaître la capacité réseau entre chacun des noeuds du cluster.

Le nouveau noeud en procédure d'adhésion doit alors successivement appeler cet algorithme et effectuer les tests jusqu'à ce que la réponse de l'algorithme indique que le graphe est complet.

Nous pouvons couper cette procédure en deux sous procédures, une montrant l'utilisation générique de l'algorithme (Alg. 1) et l'autre l'algorithme lui-même (Alg. 2).

Algorithm 1 Utilisation générale de l'algorithme

```

graph ← []
id                                ▷ Bandwidth adjacency matrix
graph[id][..] ← graph[..][id] ← 0   ▷ index of the current node in the graph
complete ← false
while complete = false do
    dest ← find_missing_bw_from(id, graph)
    if dest = -1 then
        complete ← true
    else
        bandwidth ← perfestbetween(id, dest)
        graph[id][dest] ← graph[dest][id] ← bandwidth
    end if
end while

```

Algorithm 2 Recherche d'une bande passante manquante dans le graphe des bandes passantes

```

procedure find_missing_bw_from(from, graph)
    missing ← -1
    n ← graph.len
    dest ← 0
    while dest < n do
        if from = dest then
            continue
        end if
        missing ← dest
        k ← 0
        while k < n do
            if k = from || k = dest then
                continue
            end if
            bwfk ← graph[from][k]
            bwkd ← graph[k][dest]
            if bwfk = bwkd then
                continue ▷ Cannot tell without knowing if we are in T1 or T2 situation
            else if bwfk > bwkd then
                graph[from][dest] ← bwkd
                missing ← -1
                break
            else if bwfk < bwkd then
                graph[from][dest] ← bwfk
                missing ← -1
                break
            end if
        end while
        if missing ≥ 0 then
            break
        end if
    end while
    return missing
end procedure

```

Afin de tester l'amélioration qu'apporte cette technique, nous construisons aléatoirement un nouvel arbre censé représenter un réseau commuté. Premièrement, nous générerons le corps de l'arbre qui représente le backbone du réseau et nous venons ensuite connecter des points terminaux sur les différents nœuds formant le backbone.

Les tests sont produits en fonction du nombre de nœuds qui constituent le backbone ainsi que les points terminaux. Cela permet de voir l'évolution du nombre de tests nécessaires par point terminaux en fonction de la taille du réseau. Dans les premiers diagrammes, les tests sont exécutés avec 1000 valeurs de bande passante possibles pour chaque arête du graphe. Cela correspond à prendre en compte tous réseaux ayant des liens allant de 0 à 1 Gbit/s.

Le diagramme suivant (Figure 4) montre le nombre de tests nécessaires à exécuter en moyenne à chaque ajout d'un nouveau nœud dans le cluster. La moyenne est définie sur la base de 100 graphes différents par taille de cluster. La taille du cluster indiqué sur l'axe des abscisses montre le nombre de points terminaux. Il faut savoir qu'en plus de la taille indiquée, le backbone contient toujours la moitié des nœuds terminaux. Donc en réalité, le graphe représentant le réseau pour un cluster de 60 nœuds contient 90 nœuds réseau, 60 étant des terminaux et 30 étant des commutateurs. Les barres d'erreur supérieures et inférieures montrent le maximum (respectivement le minimum) du nombre moyen de tests par noeud dans les 100 simulations.

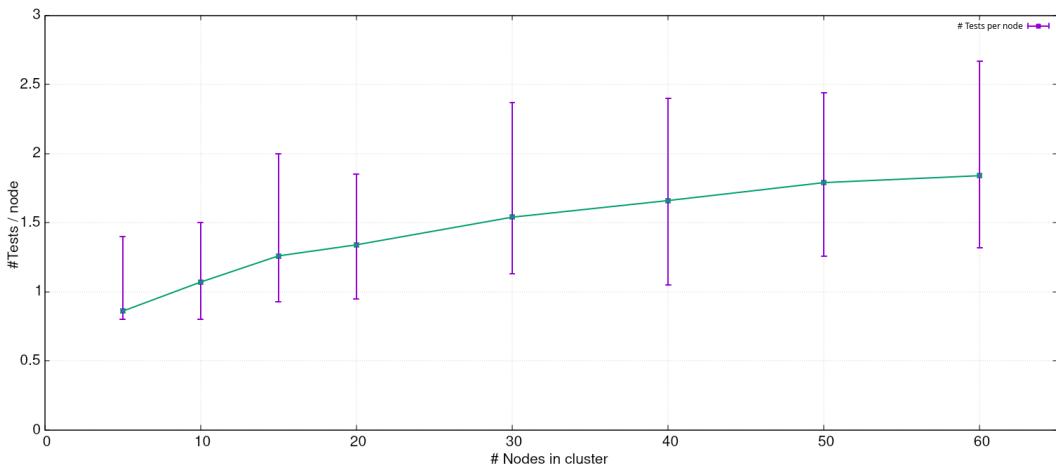


Figure 4: Nombre de tests par nœud nécessaires en fonction du nombre de nœuds dans le cluster.

La Figure 4 montre que la taille du cluster n'a que très peu d'impact sur le nombre moyen de tests à exécuter. Nous nous trouvons dans une croissance sous-linéaire.

Nous notons aussi que les résultats montrent un très grand gain en nombre de tests à exécuter. Effectivement, dans le graphique précédent, nous pouvons voir que dans le pire des cas lors de la simulation, il a fallu faire en moyenne de 2.67 tests par nœud. Ce résultat est nettement inférieur à celui donné par l'algorithme naïf qui demande pour $n = 60$ de procéder à 1770 tests, soit une moyenne de 29.5 tests par noeuds.

Comme indiqué en dessus, les tests sont exécutés avec des bandes passantes sur chacune des arêtes pouvant aller de 0 à 1000 Mbits/s. Le graphique ci-dessous (Figure 5) montre l'influence de la variation de cette limitation pour un cluster de 20 noeuds. Nous pouvons remarquer que le changement de la plage de bande passante n'a pas d'effet sur l'efficacité de l'algorithme.

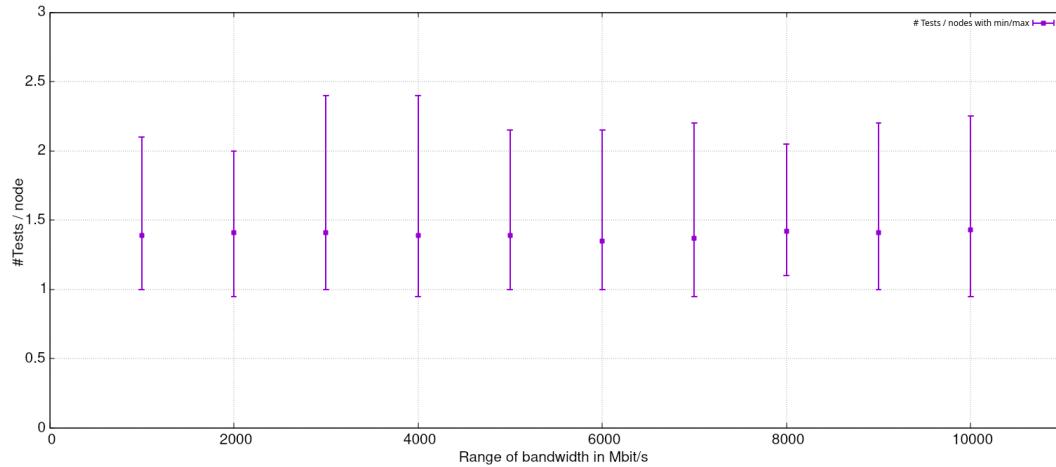


Figure 5: Nombre de tests par nœud en fonction de la plage de la bande passante

Finalement, pour mieux visualiser le réseau produit par génération automatique, le graphe ci-dessous (Figure 6) montre le réseau utilisé pour l'obtention du meilleur résultat lors de la simulation avec 60 nœuds terminaux. Le résultat obtenu était de 1.32 tests en moyenne par nœud. Les liens en rouge montrent les liens entre les commutateurs et représentent le backbone. En noir, il s'agit des connexions de nœuds terminaux. Les nombres présents sur les liens représentent le débit en Mbit/s de ces liens.

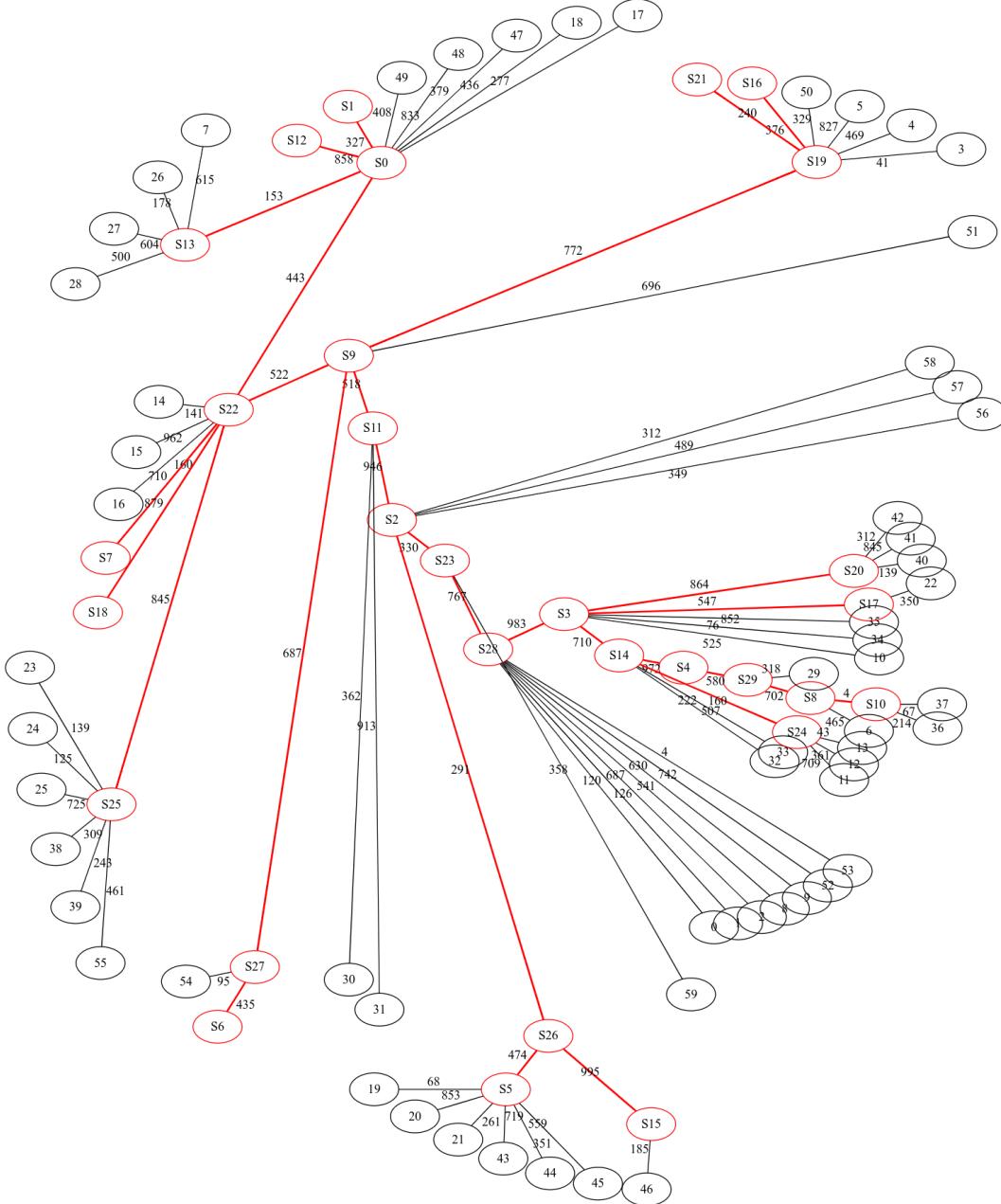


Figure 6: Graphe de 60 nœuds terminaux dans un réseau commuté.

Ce chapitre conclut l'analyse liée à la création d'un cluster Kollaps. Nous savons maintenant de quoi est constitué le cluster et quels outils peuvent être utilisés dans ce projet ou dans l'avenir. Nous avons aussi une technique de limitation des tests de performance pour atteindre des objectifs de temps viables lors la modification de la taille du cluster.

Il s'agit maintenant, pour la deuxième partie du projet, de pouvoir répartir la charge de travail sur les différentes machines qui forment le cluster Kollaps tout en respectant les contraintes demandées par les expérimentations.

3.8 Répartition des émulations

Le but de ce projet est de supporter un changement d'infrastructure dynamique. Accepter des nouvelles machines dans un cluster et ne pas s'en servir est inutile. Il faut pouvoir répartir une expérimentation sur ces différentes machines qui forment le cluster Kollaps.

Une expérimentation est constituée de processus terminaux que nous nommons PT et qui sont liés entre eux par un réseau émulé. Les capacités de connexion entre ces PTs sont donc directement liées au réseau émulé, et ce sont ces capacités qui sont modulées grâce à l'utilisation du linux traffic control. Par exemple, la Figure 7 ci-dessous montre le réseau décrit par une expérimentation. En rouge avec une bordure pleine, nous voyons les éléments qui sont lancés sur des nœuds Kollaps et en vert traitillé, les éléments qui sont émulés via l'utilisation du linux traffic control.

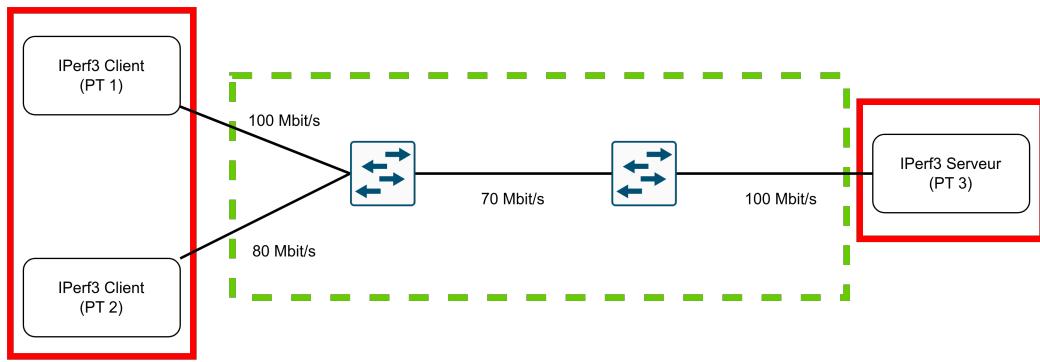


Figure 7: Exemple d'une expérimentation avec trois processus terminaux et un réseau émulé contenant 2 ponts

Pour répartir cette expérimentation sur trois machines du cluster Kollaps (un PT par machine), que nous nommons M1, M2 et M3 pour celle qui héberge PT1, respectivement PT2 et PT3, nous devons nous assurer que les connexions entre les machines soient suffisantes.

- a) $M1 \sim M2 : 80 \text{ Mbit/s}$
- b) $M1 \sim M3 : 70 \text{ Mbit/s}$
- c) $M2 \sim M3 : 70 \text{ Mbit/s}$

Dans le cas où il est impossible de trouver un ensemble de machines dans le cluster Kollaps qui répondent à ces critères, il est encore possible de déployer plusieurs PTs sur une seule machine.

La Figure 8 présente un exemple de déploiement possible correspondant à l'expérimentation montrée dans la Figure 7 sur un cluster Kollaps contenant quatre nœuds (M1 à M4) représentés sur le réseau physique.

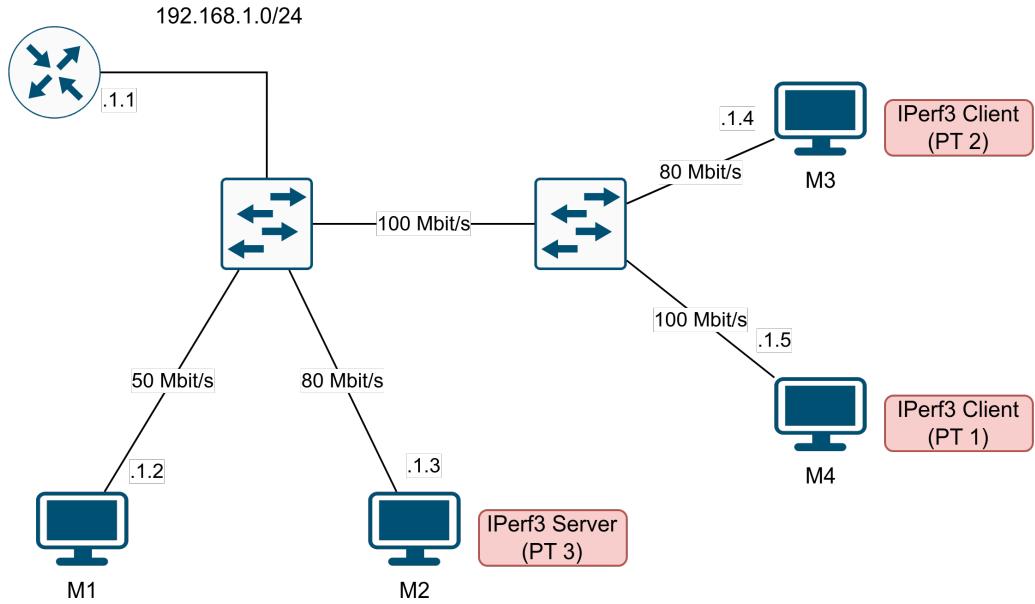


Figure 8: Déploiement possible de l'expérimentation sur la Figure 7 sur un cluster Kollaps de 4 nœuds

De manière générale, pour éviter de surcharger un nœud, il faut essayer de répartir au maximum chaque expérimentation. Il faut essayer de trouver une correspondance avec le réseau du cluster Kollaps et le réseau de l'expérimentation qui doit être émulé. Si aucune correspondance ne peut être trouvée, il est possible de considérer le déploiement de deux PTs sur un même nœud, puis trois, etc. jusqu'à trouver une correspondance. Dans le pire des cas, tous les PTs sont déployés sur le même nœud Kollaps.

Au long du projet, nous appelons le fait de faire cette distribution de charge, l'orchestration. Nous parlerons plus loin d'orchestrateur. Il s'agit de l'élément en charge de faire cette distribution.

Il existe plusieurs techniques permettant d'allouer des ressources virtuelles à des ressources physiques. C'est le problème que nous devons résoudre.

3.8.1 Technique choisie

Suite aux conseils de Dr. Guilherme Piégas Koslovski, spécialisé dans le domaine, la technique d'allocation retenue se base sur le problème d'isomorphisme de graphes.

L'isomorphisme entre deux graphes est une fonction de bijection entre les sommets du premier et du deuxième. Cela signifie qu'il faut trouver une fonction qui attribut pour chacun des sommets du premier graphe un sommet sur le deuxième graphe tel que les arêtes sont respectées. Par exemple, si deux sommets $\{u, v\}$ dans le premier graphe sont attribués aux sommets $\{x, y\}$ dans le deuxième, alors s'il existe une arête entre u et v , il doit exister une arête entre x et y .

Dans notre cas, les deux graphes sont :

- La matrice des bandes passantes des nœuds du cluster Kollaps.
- La matrice des bandes passantes des PTs de l'expérimentation.

Dans les deux cas, ces matrices représentent des graphes complètement connectés et donc il existe une arête entre chacun des nœuds. Ce que nous devons faire, c'est s'assurer que chaque arête dans le graphe de destination (ici le cluster Kollaps) indique une bande passante supérieure ou égale au graphe de départ (ici le graphe d'expérimentation).

Dans le cas de la recherche d'isomorphisme, il y a toujours un PT de l'expérimentation par sommet du graphe du cluster Kollaps. S'il est nécessaire, il est possible de dédoubler les sommets dans le graphe du cluster.

Pour trouver un isomorphisme entre deux graphes, il faut qu'ils aient la même taille. C'est pourquoi nous devons faire de l'isomorphisme de sous-graphes. Cela signifie que si le graphe du cluster Kollaps est plus grand que celui de l'expérimentation, nous devons sélectionner uniquement un sous-ensemble des sommets du graphe du cluster Kollaps. À l'inverse, lorsque le graphe d'expérimentation est plus grand, nous devons dupliquer les nœuds du cluster Kollaps. Si nous trouvons un isomorphisme en dupliquant des nœuds, cela signifie simplement que plusieurs PTs doivent être déployés sur un même nœud du cluster Kollaps.

Afin de sélectionner un sous-graphe, il est possible de générer toutes les combinaisons possibles des sommets d'un graphe. En essayant les combinaisons les unes après les autres jusqu'à trouver un sous-graphe qui convient.

Si aucun sous-graphe ne convient, il faut donc modifier l'offre, c'est-à-dire ajouter des sommets dans le graphe du cluster Kollaps en dupliquant certains sommets. Cela a pour effet de permettre la considération d'un déploiement utilisant plusieurs fois le même nœud Kollaps pour plusieurs PTs. Il faut ensuite recommencer l'étape de sélection de sous-graphes.

En faisant cette modification du graphe du cluster Kollaps itérativement, l'orchestrateur essaie toujours de répartir au maximum sur les nœuds existants puis se rapproche de plus en plus vers un déploiement sur un seul nœud du cluster.

Il pourrait sembler qu'il suffit de trouver un isomorphisme de graphe entre l'expérimentation et le cluster Kollaps qui vérifie uniquement si la bande passante du cluster est plus grande que celle demandée par l'expérimentation. Mais d'autres problèmes peuvent intervenir en fonction des expérimentations. C'est pourquoi, pour assurer que le réseau sous-jacent est complètement capable de supporter une expérimentation, il faut que l'orchestrateur soit prudent.

3.8.2 Attribution conservatrice

L'isomorphisme de graphe est une très bonne technique pour parvenir à nos fins. Cependant, il y a une chose qui doit être prise en compte. Nous ne connaissons pas le réseau sous-jacent, mais uniquement la bande passante entre les différents nœuds Kollaps. Il n'est donc pas possible de savoir quels liens sont partagés sur le réseau sous-jacent. Kollaps connaît la bande passante uniquement lorsqu'elle est utilisée par un seul nœud lors d'un test de performance.

Si nous prenons par exemple l'expérimentation donnée par le graphe ci-dessous (Figure 9) :

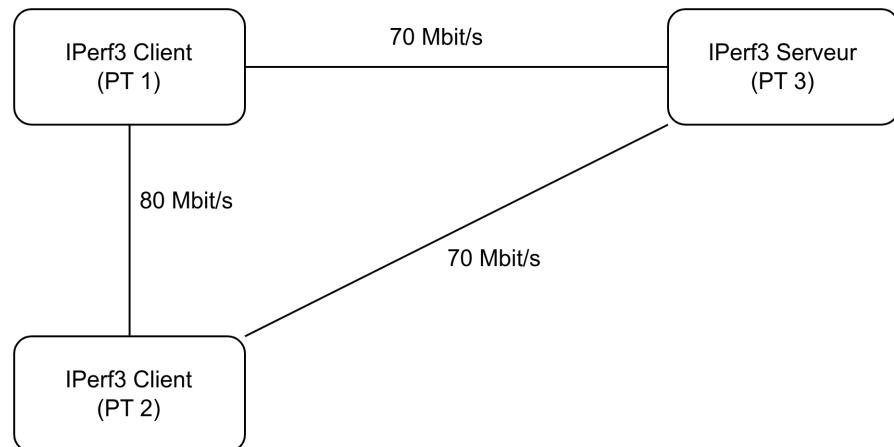


Figure 9: Expérimentation Kollaps

Cette expérimentation implique que deux flux de 70 Mbit/s puissent partir de PT1 et de PT2 simultanément en direction de PT3, sans devoir diviser de la bande passante.

La matrice de bande passante est exactement la même que pour l'expérimentation présentée précédemment sur la Figure 7. Cependant, il y a une grosse différence. Dans l'expérimentation présentée précédemment, PT1 et PT2 partagent les mêmes liens entre eux et PT3. Cela signifie que si les deux utilisent le maximum du réseau émulé, ils ont tous les deux (PT1 et PT2) droit à 35 Mbit/s pour atteindre PT3. Pour la deuxième expérimentation, ce n'est pas le cas. Les deux doivent avoir 70 Mbit/s de débit vers PT3.

Cela implique que la distribution présentée ci-dessous (Figure 10) est correcte pour la première expérimentation, mais fausse pour la seconde.

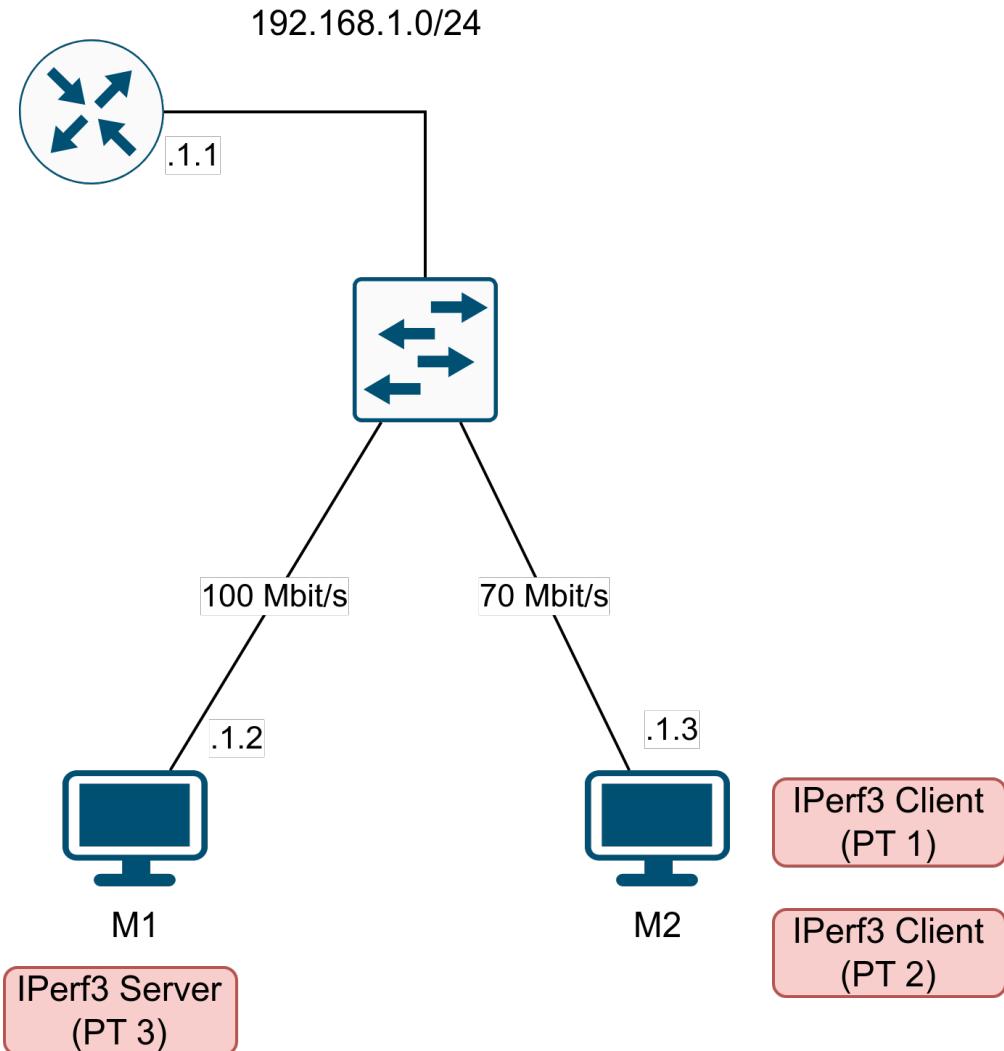


Figure 10: Déploiement non valide d'une expérimentation Kollaps dû à la distribution conservatrice.

Pour la première expérimentation avec le partage de lien, la capacité du lien 70 Mbit/s va aussi être divisé et suivre l'émulation. Par contre, pour la deuxième expérimentation, nous avons besoin en réalité, sur le réseau sous-jacent, que le lien de 70 Mbit/s soit de 140 Mbit/s pour assurer les deux connexions simultanées de 70 Mbit/s.

L'orchestrateur doit être au courant de cela et, à défaut de connaître réellement le réseau sous-jacent, doit considérer que toutes les connexions passeront par un même lien sur le réseau physique (pire des cas). Il requiert donc pour la distribution que la bande passante entre deux nœuds Kollaps M1 et M2 soit au minimum égale à la somme des bandes passantes de tous les PTs prévus sur M1 vers tous les PTs prévus sur d'autres machines.

C'est ce qui est appelé dans ce projet : la distribution conservatrice.

Cette distribution conservatrice est basée sur le pire des cas. Elle est nécessaire pour garantir la connexion demandée par l'expérimentation quand celle-ci contient différents chemins pour accéder à un même PT.

Le problème apparaît, car nous ne connaissons pas la topologie du réseau sous-jacent.

Ce projet utilise la technique de la distribution conservatrice. Pour de futures améliorations, il serait possible d'implémenter différentes solutions permettant de limiter l'effet de ce problème. Premièrement, donner la possibilité à l'administrateur de fournir une carte de topologie réseau en indiquant par quelque moyen quel nœud Kollaps se connecte sur quel élément réseau. La Figure 11 suivante montre une possibilité de mise en place en utilisant des intervalles IPs comme identifiants. La capacité du lien entre un nœud et l'appareil réseau avec lequel il est directement connecté peut ensuite être découverte sans test via le protocole d'auto-négociation Ethernet. Une deuxième solution serait de mettre en place une découverte réseau automatique comme discutée dans le chapitre 3.5.

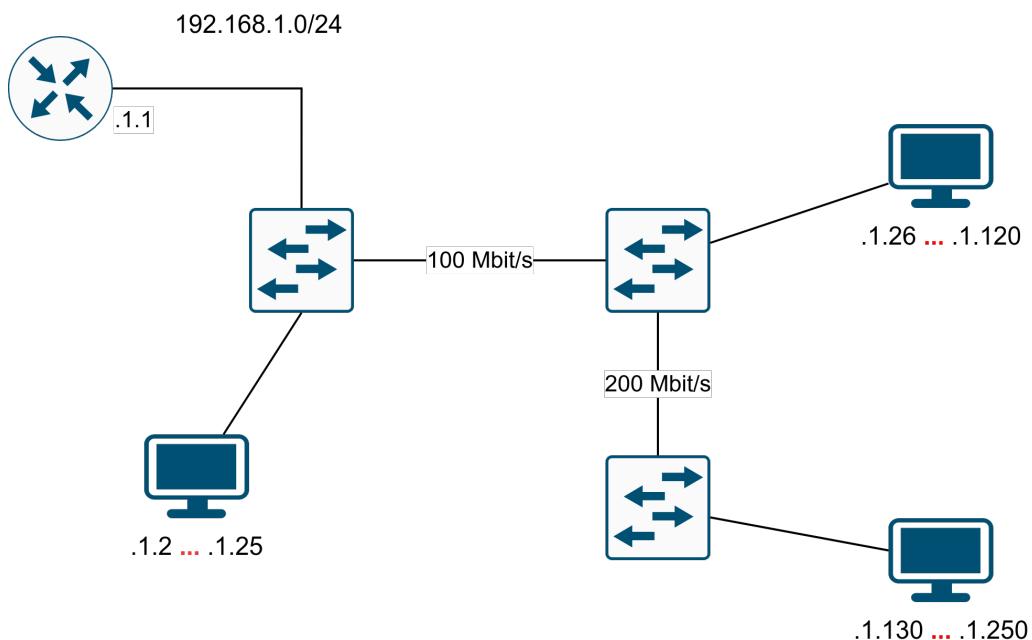


Figure 11: Topologie réseau définie par l'utilisateur en utilisant des intervalles IPs comme identifiants

Lors de la distribution d'une expérimentation sur plusieurs machines physiques, le terme partie d'expérimentation est utilisé pour décrire un groupe de processus terminaux issus de cette même expérimentation qui se trouve sur une machine du cluster. Alors, l'expérimentation est divisée en parties qui contiennent chacune un ou plusieurs processus terminaux et qui sont dispersées à travers le cluster Kollaps.

Il se peut qu'un nœud supportant une ou plusieurs parties d'expérimentations décide de quitter le cluster, ou subisse un échec. C'est pourquoi il faut considérer que le cluster Kollaps n'est pas stable.

3.8.3 Tolérance aux pannes

Il existe deux cas où une expérimentation doit être annulée en cours de fonctionnement :

1. Un des nœuds Kollaps se déconnecte du cluster avec des parties d'expérimentations.
2. Une partie d'une expérimentation s'interrompt sur un des nœuds du cluster.

Dans le premier cas, il faut interrompre toutes les expérimentations qui avaient une partie distribuée sur le nœud en question. Dans le deuxième cas, il faut uniquement interrompre l'expérimentation liée à la partie qui a cessé de fonctionner.

Dans les deux cas, il est impératif que l'orchestrateur sauvegarde toutes les distributions qu'il a créées pour pouvoir diffuser un message d'annulation aux nœuds concernés si nécessaire.

L'orchestrateur joue le rôle de point de défaillance unique. Si l'orchestrateur tombe en échec, il devient impossible de détecter la terminaison de certaines expérimentations et de réceptionner les messages d'échec des nœuds du cluster. Il est impératif qu'il soit développé de manière à ne pas tomber en échec.

Une amélioration intéressante pour l'orchestrateur lors de prochains projets serait d'envisager un système de point ou de classement pour les nœuds Kollaps. Plus un nœud provoque d'échec ou se déconnecte souvent, moins il est considéré lors des recherches de distributions.

La version de Kollaps actuelle supporte plusieurs types d'expérimentation. Cette nouvelle version se focalise cependant sur un seul.

3.9 Types d'expérimentations supportés et orchestrateur

La version actuelle de Kollaps supporte actuellement deux types d'expérimentation. Le premier est le type *container* qui s'occupe de lancer les conteneurs nécessaires sur un orchestrateur à choix (Docker Swarm ou Kubernetes) puis gère les capacités réseau de ces conteneurs. Le deuxième est le type *baremetal* qui n'utilise pas d'orchestrateur, mais nécessite que les applications soient déjà actives et dans leur propre espace de nom.

Pour cette nouvelle version de Kollaps, un troisième type vient s'ajouter. Le projet se concentre principalement sur ce nouveau type.

Le nouveau type utilise aussi des conteneurs, mais cette fois n'utilise pas d'orchestrateurs existants. Kollaps est lui-même l'orchestrateur. Il s'agit d'un mélange entre les deux précédents types.

Afin de pouvoir transformer Kollaps en service, chaque nœud du cluster Kollaps doit être prêt à recevoir une nouvelle émulation à exécuter. Pour ce faire, il est nécessaire d'avoir un processus qui attend ces requêtes sur la machine. Ce processus est directement lancé sur la machine et non dans un conteneur. Il s'agit là de la partie *baremetal*. Le processus est l'application Kollaps.

Lorsqu'un nœud du cluster Kollaps reçoit une partie d'expérimentation à émuler, il s'occupe de lancer des conteneurs pour chacun des processus terminaux (PTs) qu'il doit exécuter pour cette partie d'émulation. Il s'agit de la partie *container*.

Il est tout à fait possible dans l'avenir d'utiliser la même infrastructure pour implémenter la technique *baremetal* originelle. Il suffit d'implémenter dans le langage de description de topologie et dans l'orchestrateur le fait de verrouiller une application pour un nœud spécifique sur lequel l'application en question est déployée.

Les expérimentations Kollaps ont toujours eu pour but d'être le plus décentralisées possible. Avec ce nouvel orchestrateur, cet objectif ne change pas. Comme les expérimentations sont réparties à travers le cluster Kollaps, il faut un moyen pour que chacune des parties d'une expérimentation puisse se synchroniser.

3.10 Synchronisation entre parties d'expérimentation

Une expérimentation ne décrit pas uniquement le graphe à émuler, mais aussi différents événements qui peuvent intervenir. Lors de la distribution d'une expérimentation, chaque nœud Kollaps reçoit la liste des processus terminaux qu'il doit démarrer / montrer ainsi que la liste des événements de l'expérimentation. De cette manière, chaque nœud participant à l'émulation d'une expérimentation applique seul les événements.

Les événements sont définis par rapport au début de l'expérimentation en termes de temps. Cela signifie que toutes les parties d'une expérimentation doivent se synchroniser pour commencer l'expérimentation en même temps.

Afin de pouvoir se baser sur le temps de la machine, il faut que celles-ci soient synchronisées entre elles.

Plusieurs algorithmes ou techniques existent pour ceci. Dans les plus connus, se trouvent l'algorithme de Berkeley et la synchronisation via NTP (Network Time Protocol).

L'algorithme de Berkeley désigne un leader au sein d'un cluster de machines qui sert de source du temps et qui se base sur le *Round Trip Time* (RTT) avec les autres machines pour leur fournir un temps ajusté en fonction de la connexion réseau qui les sépare [8]. Cet algorithme peut être un excellent choix pour le cluster Kollaps car nous possédons déjà une infrastructure avec leader. Il est donc possible d'intégrer cet algorithme au nouvel orchestrateur, ce qui a pour avantage de ne pas dépendre de la configuration de chacune des machines.

Une autre solution à la synchronisation du temps est l'utilisation du NTP. Le protocole est intégré dans chacun des systèmes d'exploitation courants, à savoir macOS, Linux et Windows. Se baser sur une synchronisation par NTP implique que chacune des machines doit être configurée correctement avant son utilisation dans un cluster Kollaps. L'utilisation du NTP donne plus de responsabilité à l'utilisateur final.

En considérant que les machines ont une heure synchronisée via une des techniques (ou autres) expliquées ci-dessus, il reste à résoudre la synchronisation de l'heure de départ pour une expérimentation.

La prise de décision se déroule dans un environnement ayant plusieurs caractéristiques importantes :

1. Chaque nœud est initialisé lors de la prise de décision.
2. Un des nœuds participant à une expérimentation peut avoir un problème, ou s'être déconnecté du cluster, ce qui le rend inaccessible.
3. Si un nœud est actif, il est loyal (dans le sens décrit par le problème des généraux Byzantins [10]).

Ce chapitre aborde deux façons de résoudre le problème : centralisée ou décentralisée.

Une façon centralisée d'aborder ce problème serait qu'un nœud leader d'une expérimentation envoie une proposition d'heure de départ à tous les autres nœuds. Une fois la proposition envoyée, il attend pour les différents accusés de réception. Une fois les réponses récoltées, il envoie un signal de confirmation à tous les nœuds et attend à nouveau un accusé de réception par nœud. Si la procédure se termine avant l'heure proposée initiale, alors nous sommes garantis que l'heure de départ utilisée par les différents nœuds est la même. Dans le cas contraire, le leader peut recommencer la procédure si l'heure initiale a été dépassée avant l'envoi du premier signal de confirmation ou doit annuler l'expérimentation si au moins un signal de confirmation a été reçu par un nœud non-leader. Cette technique oblige d'envoyer $4 \cdot (n - 1)$ messages pour une expérimentation distribuée sur n nœuds, dû au fait de devoir réceptionner des accusés de réception.

Une seconde approche, elle, décentralisée, se base sur le fait que chaque nœud est responsable du suivant. Un ordre partiel entre les nœuds peut être défini sur la base de leur adresse IP. Ceci permet de définir pour chaque nœud un précédent et un suivant. L'algorithme utilise cette propriété pour définir un système de synchronisation. La procédure se représente sous forme de cercle où un jeton est passé d'un noeud à un autre. L'approche présentée subit les mêmes contraintes en terme d'annulation d'une expérimentation et d'adaptation à la volée de l'heure de départ que l'approche centralisée. Cependant, elle permet d'éliminer l'utilisation des accusés de réception, ce qui diminue le nombre de messages à transmettre.

L'algorithme est le suivant : un leader (définit par le fait qu'il est le premier nœud) lance la procédure de synchronisation en envoyant une proposition d'heure de départ au suivant. En même temps, il déclenche un chronomètre. Lorsqu'un nœud reçoit une proposition d'heure, il la passe au nœud suivant. Quand la proposition arrive à nouveau chez le leader, celui-ci arrête son chronomètre. Il a alors la durée approximative que prend le passage d'une information à travers les nœuds du cluster. S'il reste suffisamment de temps pour refaire un tour des nœuds avant d'atteindre l'heure de début (en prenant une marge), il lance alors un signal de confirmation. Dans le cas contraire, il envoie une nouvelle proposition d'heure de commencement qui est supérieure à deux fois le temps pris pour faire le tour des nœuds. Lorsqu'un nœud reçoit un signal de confirmation, il vérifie la validité de ce signal. La validité d'un signal de confirmation est définie par l'heure actuelle. Si la proposition que confirme ce signal indique une heure passée ou l'heure actuelle, le signal n'est plus valide. Dans le cas contraire, il est valide. Une fois que le signal de confirmation retourne au leader, celui-ci sait que l'expérimentation est synchronisée.

Le fonctionnement optimal est représenté sur la Figure 12, où les flèches bleues montrent le passage d'une proposition et les flèches vertes montrent le passage du signal de confirmation.

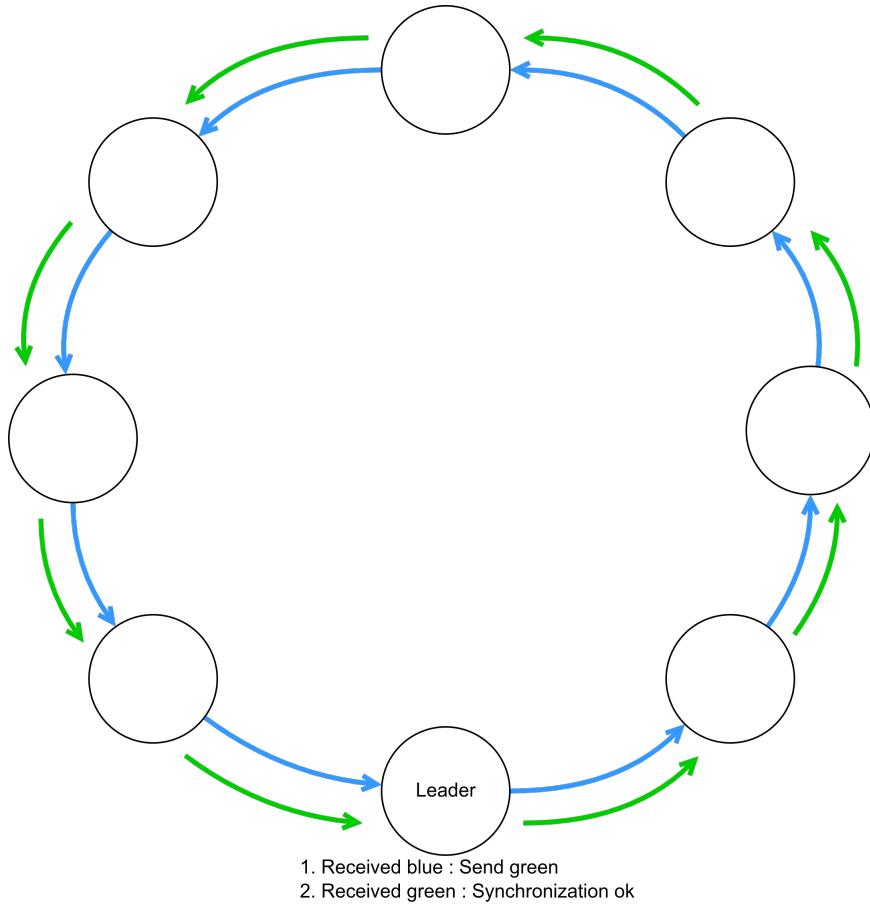


Figure 12: Procédure de synchronisation de l'heure de départ d'une expérimentation

Dans le cas où un signal de validation n'est pas valide, nous arrivons dans une situation où, comme certains nœuds ont déjà reçu le signal, certaines parties de l'expérimentation ont déjà commencé. Cette situation est une désynchronisation. Le nœud qui détecte la désynchronisation est responsable d'envoyer un message d'annulation à une entité externe pour qu'elle puisse annuler les parties de l'expérimentation en question sur les différents nœuds. L'entité externe, dans notre projet, est l'OManager (voir ch. 4.6).

La Figure 13 représente le phénomène de désynchronisation.

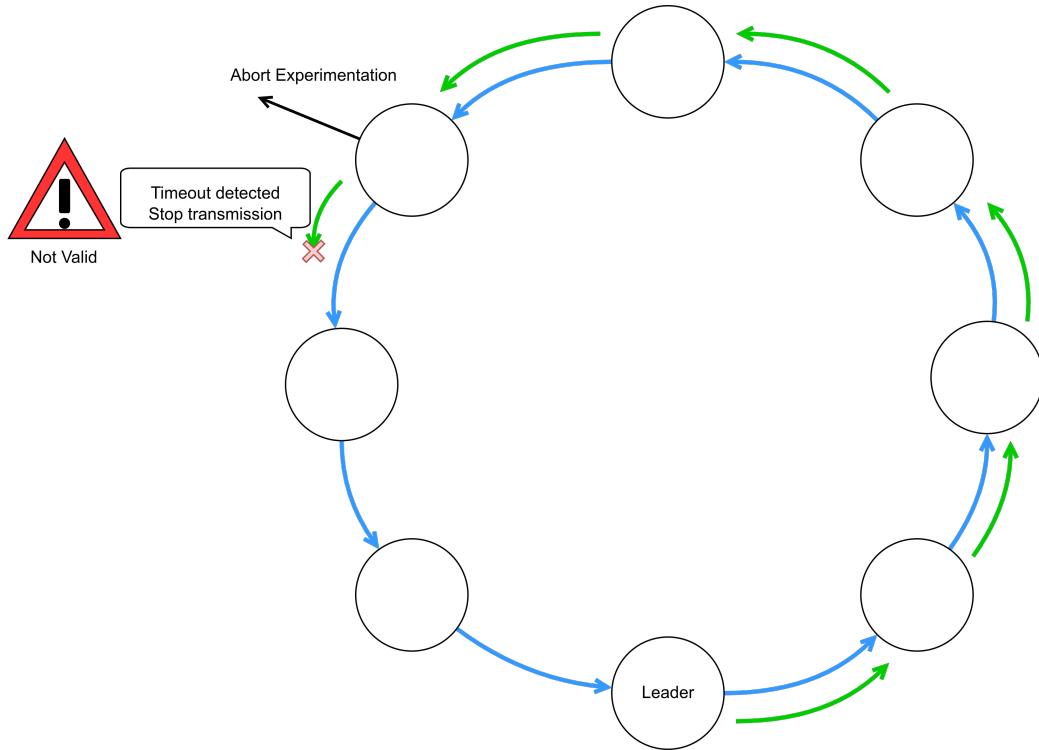


Figure 13: Phénomène de désynchronisation

Il se peut qu'un nœud ne soit plus atteignable et que sa sortie du cluster Kollaps n'ait pas encore été détectée. C'est la situation que représente la Figure 14. Dans ce cas, un nœud ne va pas être capable de contacter son suivant. Lorsque cela arrive, c'est le nœud qui détecte une erreur de liaison qui envoie un signal d'annulation, et non le leader. Il se peut, dans un système à deux nœuds, que le leader envoie un signal de confirmation et tombe en échec instantanément. Le deuxième nœud ne peut donc pas communiquer à nouveau avec le leader, mais doit quand même signaler l'annulation de l'expérimentation à l'entité externe. C'est pourquoi le nœud qui détecte une erreur de connexion est le lanceur du signal d'annulation.

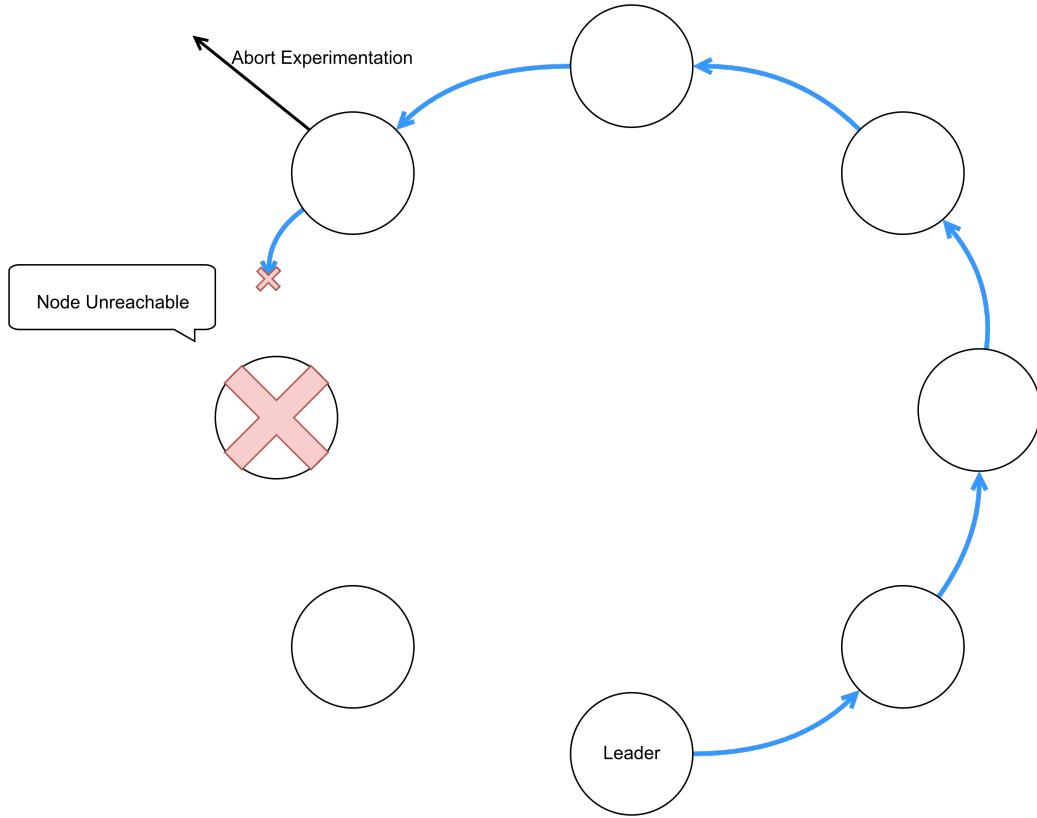


Figure 14: Nœud inaccessible lors de la synchronisation

3.11 Traffic Control AL et adressage

Kollaps utilise une librairie nommée Traffic Control Abstraction Layer (appelée ensuite TCAL). Cette librairie écrite en C permet de modifier facilement les valeurs du linux traffic control. Elle est reprise dans cette nouvelle version.

Le TCAL permet uniquement de modifier les valeurs du linux traffic control en direction d'une destination définie par une adresse IP. Cela requiert que si plusieurs PTs sont déployés sur une seule machine, qu'ils aient tous une adresse IP différente. Sinon, une limitation via le linux traffic control n'affectera pas seulement la connexion entre un PT source et un PT destination, mais entre le PT source et tous les PTs se trouvant sur le même nœud que celui de destination.

Une solution simple pour enlever ce problème serait d'appliquer les paramètres du linux traffic control non seulement à destination d'une adresse IP, mais en fonction d'une adresse et d'un port. Ceci permettrait que plusieurs PTs partagent une même adresse avec chacun un port différent. Cependant, cela amène une autre limitation qui va à l'encontre du principe de Kollaps. La notion de port n'est pas présente dans l'Internet Protocol. Cette notion est uniquement présente dans certains protocoles comme l'UDP et le TCP. L'utilisation du port limite donc les PTs à une utilisation des protocoles UDP et TCP. Kollaps se veut être transparent sur un réseau. Il ne veut pas limiter les utilisateurs à l'utilisation de certains protocoles.

Une autre solution serait d'implémenter son propre protocole que nous nommons KP pour Kollaps Protocol. Cette solution est discutée dans cette thèse, mais n'est pas implémentée pour cause de temps. Le KP est une couche se trouvant directement entre la couche IP

et la couche de transport. Son but est d'encapsuler des paquets avec l'identifiant du PT de destination. Cette technique demande de transformer à la volée les paquets sortants d'un espace de nom pour substituer le protocole de transport utilisé par le protocole KP. La transformation de paquets à la volée est faisable grâce à des programmes eBPF[16].

Un paquet sortant doit être modifié avant qu'il soit intercepté par le linux traffic control, de cette manière, il est possible d'appliquer des configurations TC uniquement sur les paquets utilisant le Kollaps Protocol. Un autre programme eBPF pourrait, lui, à la détection d'un paquet entrant utilisant le KP, transformer le paquet pour supprimer cette couche supplémentaire. La suppression de la couche permet alors de supprimer les modifications apportées et de rendre ce travail invisible aux yeux des processus déjà en place. Cette technique apporte un autre problème.

Kollaps veut limiter l'utilisation du réseau uniquement envers les PTs qui font partie de l'expérimentation. Il se peut qu'un PT fasse une requête sur un site quelconque. Dans ce cas, le paquet ne doit pas être modifié. La modification du paquet doit uniquement intervenir lorsqu'il est destiné à un PT de l'expérimentation. Il faut donc que le programme de transformation ait une fonction qui transforme l'adresse IP de destination à un identifiant. Si l'adresse IP n'est pas gérée par Kollaps, alors le paquet ne subi pas de transformation. Le but est de limiter l'utilisation d'adresses IP supplémentaires. Si nous devons attribuer une adresse IP à chaque PT afin de pouvoir faire la transformation, quel est le but ? Le principe est de ne pas attribuer une adresse réelle à chaque PT mais de leur attribuer une adresse provenant d'un sous-réseau fictif. De cette manière, lorsque deux PTs souhaitent s'échanger des messages, ils le font en utilisant leur adresse fictive, mais en réalité, le paquet est modifié pour être redirigé vers l'adresse du nœud Kollaps qui héberge le PT de destination.

Cette idée de surcouche rejoint l'idée d'un Service-Mesh, mais uniquement pour les PTs de Kollaps.

La Figure 15 ci-dessous montre le principe de fonctionnement d'un tel système.

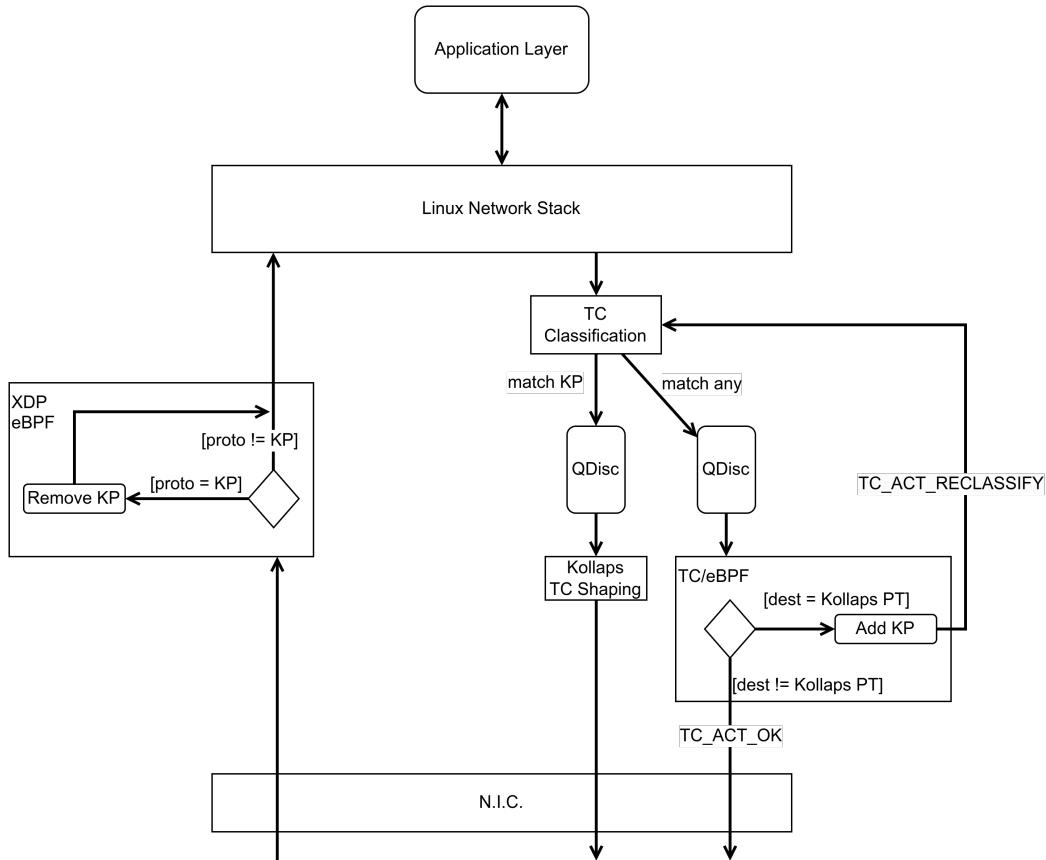


Figure 15: Idée d'implémentation du protocole Kollaps

La technique est présentée comme une idée. Cette proposition n'a pas fait l'objet d'investigations plus poussées, ni de réels tests. Il s'agit néanmoins d'une piste pour améliorer Kollaps. Ci-dessous une liste de références sur lesquelles se base la proposition :

- [XDP et eBPF Tutorial⁹](https://www.tigera.io/learn/guides/ebpf/ebpf-xdp/) : Tigera
- [eBPF: When \(and when not\) to use it¹⁰](https://www.tigera.io/blog/ebpf-when-and-when-not-to-use-it/) : Tigera
- [eBPF an overview¹¹](https://cloudyuga.guru/hands_on_lab/ebpf-intro) : CloudYuga
- [eBPF and the Service Mesh: Don't Dismiss the Sidecar Yet¹²](https://www.infoq.com/articles/ebpf-service-mesh/) : InfoQ
- [Differentiate three types of eBPF redirects¹³](http://arthurchiao.art/blog/differentiate-bpf-redirections/#3-process-redirect-logic-in-tc-bpf) : Arthur Chiao
- [Understanding tc "direct action" mode for BPF¹⁴](https://quentinmonnet.com/post/understanding-tc-direct-action-mode-for-bpf/) : Quentin Monnet
- [Linux Network Stack Walkthrough \(2.4.20\)¹⁵](https://qmonnet.github.io/whirl-offload/2020/04/11/tc-bpf-direct-action/) : jsevy
- [Linux Networking¹⁶](https://linux-kernel-labs.github.io/refs/heads/master/labs/networking.html#) : Linux Kernel Lab

⁹<https://www.tigera.io/learn/guides/ebpf/ebpf-xdp/>

¹⁰<https://www.tigera.io/blog/ebpf-when-and-when-not-to-use-it/>

¹¹https://cloudyuga.guru/hands_on_lab/ebpf-intro

¹²<https://www.infoq.com/articles/ebpf-service-mesh/>

¹³<http://arthurchiao.art/blog/differentiate-bpf-redirections/#3-process-redirect-logic-in-tc-bpf>

¹⁴<https://qmonnet.github.io/whirl-offload/2020/04/11/tc-bpf-direct-action/>

¹⁵https://jsevy.com/network/Linux_network_stack_walkthrough.html

¹⁶<https://linux-kernel-labs.github.io/refs/heads/master/labs/networking.html#>

- [BPF: Using BPF to do Packet Transformation¹⁷](#) : Oracle - Alan Maguire
- [Packet Flow in Netfilter and General Networking¹⁸](#) : Wikipédia Authors
- [Modify UDP packet using eBPF¹⁹](#) : taoshu
- [The u32 classifier²⁰](#) : Bert Hubert and others.

Dans ce projet, nous considérons que chaque PT a sa propre adresse. L'adresse est donnée par l'utilisateur lors de la définition d'une expérimentation. Si un conteneur doit être déployé par un des nœuds Kollaps, il utilise le réseau IpVlan de Docker qui permet de connecter des conteneurs sur le réseau tout en leur attribuant une adresse IP[3]. L'IpVlan agit pour le conteneur comme s'il était directement sur le réseau.

Ce dernier chapitre conclut l'analyse du projet. Le projet bénéficie des notions développées lors de cette analyse et implémente les différentes techniques et les algorithmes développés. Les chapitres suivants sont dédiés à la conception du nouveau système Kollaps.

4 Conception

Les points centraux de cette nouvelle version de Kollaps sont la création d'un cluster de machines ainsi que l'orchestration et la répartition des expérimentations. La conception du logiciel est basée sur ces objectifs. Elle permet de répondre aux critères demandés en définissant un logiciel constitué de plusieurs composants, chacun avec des rôles précis.

Les chapitres suivants présentent tout d'abord une vue générale de l'infrastructure de l'application. Ensuite, pour chacun des composants majeurs, certaines de leurs propriétés sont expliquées plus en détails. Ces chapitres n'ont pas pour but d'expliquer le fonctionnement complet de chacun des composants, mais plutôt de documenter certains points importants.

Certaines explications font référence à des structures qui ne sont pas forcément décrites. Si c'est le cas, c'est que le nom de la structure est suffisamment explicite pour comprendre son utilité. Un diagramme de classe complet des structures indépendantes est disponible en annexe au chapitre 11.4.

Il faut avoir conscience que le langage utilisé est Rust. Rust n'est pas un langage orienté objet. C'est pourquoi, certains diagrammes ne peuvent pas s'appliquer directement à l'implémentation. Les diagrammes présentés aident à la compréhension générale de la conception, mais ne peuvent pas toujours refléter exactement une implémentation en Rust.

4.1 Infrastructure

Les machines du cluster Kollaps doivent chacune participer au maintien du cluster, ainsi qu'à l'éulation d'expérimentations, ou de parties d'expérimentations. Elles partagent un ensemble de responsabilités différentes qui ont chacune leur composant dédié. Le leader a une responsabilité supplémentaire par rapport aux autres machines du cluster : il doit aussi gérer l'orchestration et la distribution des expérimentations. La Figure 16 montre les différents composants qui constituent cette nouvelle version de Kollaps.

La constitution de l'application Kollaps ne varie pas en fonction du type de machine dans le cluster (leader ou non) à l'exception de la présence du OManager qui est démarré uniquement sur la machine leader.

the-struct-sk-buff-structure

¹⁷<https://blogs.oracle.com/linux/post/bpf-using-bpf-to-do-packet-transformation>

¹⁸https://en.wikipedia.org/wiki/Express_Data_Path#/media/File:Netfilter-packet-flow.svg

¹⁹<https://taoshu.in/unix/modify-udp-packet-using-ebpf.html>

²⁰<https://tldp.org/HOWTO/Adv-Routing-HOWTO/lartc.adv-filter.u32.html>

La représentation par bloc des composants (Figure 16) se lit de bas en haut. Un bloc dépend de celui sur lequel il est reposé, sauf pour les librairies en bleue, qui sont elles indépendantes.

Le bloc central représente l'application Kollaps. Les éléments détachés sont des composants qui forment une application à part entière. Par exemple, chacun des Reporters est un processus différent.

Les librairies misent à la verticale permettent de montrer, dans leur composant respectif, quelles parties les utilisent. Typiquement, dans le composant d'émulation, cela signifie que l'EManager et les EmulCores utilisent la librairie "Docker Helper" mais que la librairie "Bandwidth Helper" est uniquement utilisée par les EmulCores.

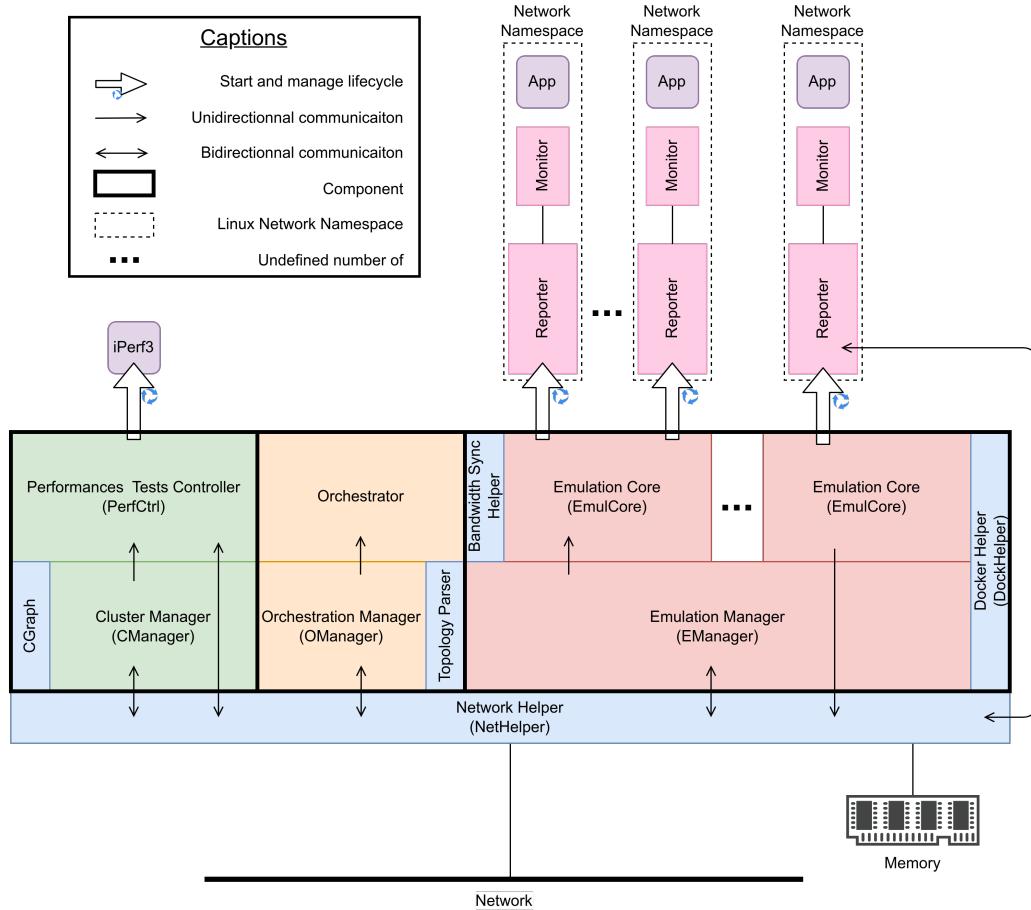


Figure 16: Composants formant l'application Kollaps

Le code couleur de lecture est le suivant :

- Vert : Éléments liés à la gestion du cluster Kollaps.
- Orange : Éléments liés à l'orchestration des expérimentations.
- Rouge : Éléments liés à l'émulation de parties d'expérimentations.
- Rose : Éléments liés à l'analyse et à la surveillance réseau de processus.
- Violet : Processus externes au système.

- Bleu : Librairies permettant d'aider les éléments dans certaines tâches.

La suite des chapitres passe à travers chacun des composants et chacune des librairies présentés pour indiquer comment est-ce qu'ils fonctionnent, leur utilité et leurs particularités. Mais d'abord, le chapitre suivant présente le fonctionnement général de cette nouvelle version de Kollaps.

4.2 Fonctionnement général et communication

Avant d'entrer dans les détails des différents composants, il est essentiel pour la compréhension d'avoir une vue d'ensemble sur le fonctionnement général de cette nouvelle version de Kollaps. Le cluster Kollaps contient un leader. Cet au leader qu'incombe la tâche d'orchestrer des nouvelles expérimentations ainsi que de gérer l'état du cluster. C'est pour cette raison qu'il y a, dans cette nouvelle version de Kollaps, une partie centralisée et une partie décentralisée.

La partie centralisée permet de gérer l'ajout et l'annulation d'expérimentations, mais aussi de gérer l'ajout et le retrait de nœuds dans le cluster Kollaps.

Comme exemple, la Figure 17 montre de manière simplifiée comment est-ce qu'une nouvelle expérimentation est déployée à travers le cluster Kollaps et quels composants communiquent pour cela.

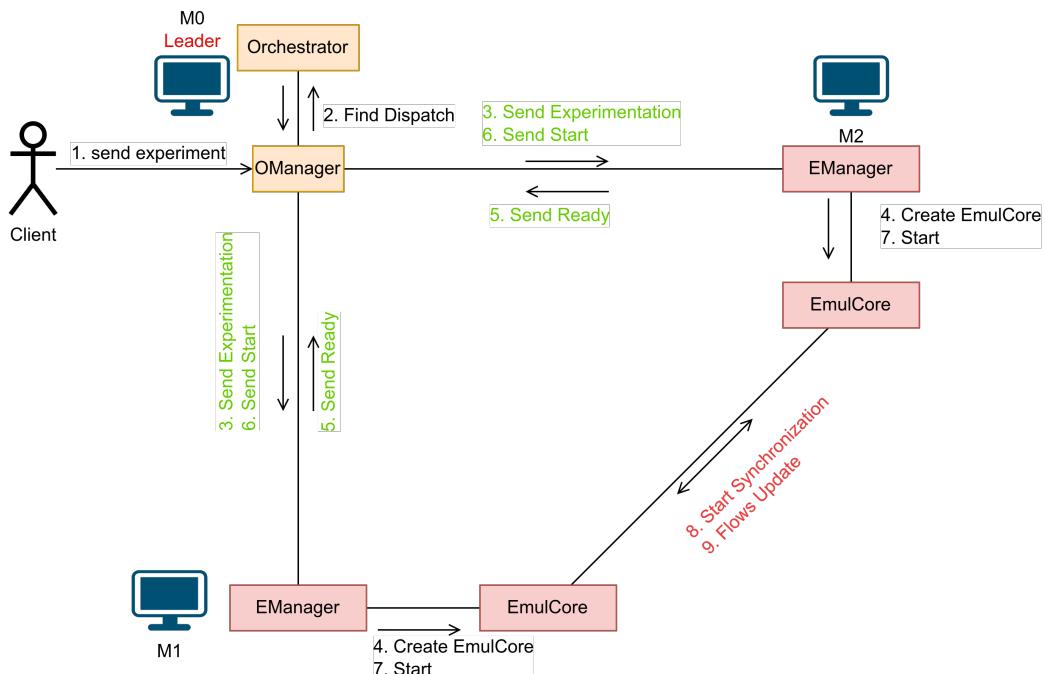


Figure 17: Communication entre composants lors d'une nouvelle expérimentation

Sur la Figure 17, les textes en vert montrent les communications qui permettent une gestion centralisée de l'expérimentation, et les textes rouges montrent les communications liées à une gestion décentralisée de chacune des parties d'une expérimentation.

Comme montré dans le chapitre précédent (ch. 4.1), chaque nœud du cluster Kollaps exécute plusieurs composants qui ont tous leurs objectifs et leur utilité. Ces composants doivent pouvoir communiquer entre eux à travers le réseau. Par exemple, les CManager

répartis sur les différentes machines doivent pouvoir s'échanger des informations par rapport à l'état du cluster, de même pour les composants d'émulation qui doivent pouvoir s'échanger des données en temps réel par rapport à une émulation.

Chaque composant qui utilise le Network Helper a un port de communication dédié. Les messages des CManagers ne se mélangent pas avec les messages des Emulation Managers par exemple. Ceci permet de garder une bonne séparation entre les composants.

4.3 NetHelper - Librairie d'aide aux communications réseau

La librairie NetHelper a pour but de simplifier les communications en réseau et locales. Elle fournit la possibilité de créer des connexions en utilisant différents protocoles, de décoder et d'encoder automatiquement des données à transmettre, ainsi que d'écouter des connexions entrantes.

Elle permet d'attacher des *Handlers* sur certains ports ou sockets en fonction d'un protocole défini. Elle est écrite de façon à pouvoir être facilement améliorée en apportant le support pour de nouveaux protocoles. Pour ce projet, elle fournit ces capacités pour trois protocoles différents : TCP, UDP et Unix pour les sockets Unix.

Cette librairie permet de faire abstraction de la partie d'encodage et de décodage des données lors de la transmission pour se concentrer uniquement sur des éléments concrets.

La Figure 18 ci-dessous montre le diagramme de classe de la librairie NetHelper. Comme nous pouvons le voir, il existe une implémentation de l'interface *Protocol* pour chacun des protocoles définis plus haut.

Une interface très importante est le *ProtocolBinding*. Cette interface décrit ce qui est possible de faire lorsque nous avons lié un protocole avec une adresse. Il est possible d'écouter, de recevoir et d'envoyer des données. D'autres méthodes spécifiques à certains protocoles sont directement créées dans le *ProtocolBinding* correspondant. Par exemple, avec UDP, nous pouvons envoyer des broadcasts sur le réseau.

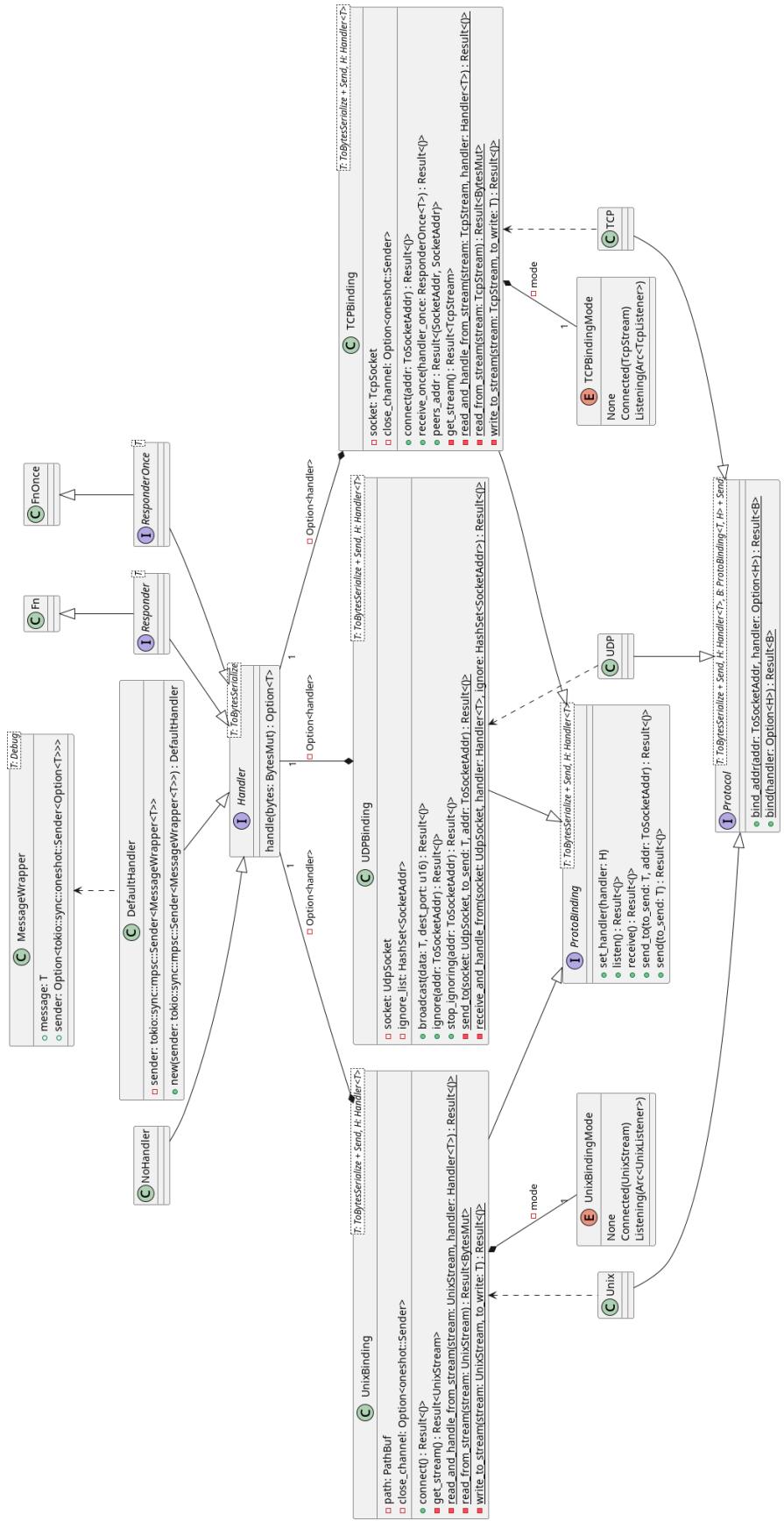


Figure 18: Diagramme de classe de la librairie NetHelper

Le concept d'Handler est un point central de la librairie. Un Handler est une structure capable de recevoir en entrée une série d'octets qui a été lue sur une connexion. Il peut ensuite effectuer une action avec ces octets et retourner un résultat. Le résultat retourné est ensuite encodé pour être transféré en réponse sur la même connexion.

Dans la librairie, il existe deux types concrets d'Handler. Le premier est le NoHandler qui ne fait rien avec les données en entrée et ne répond rien. Il est utile lorsque que l'on veut juste créer une connexion et envoyer des données sans se préoccuper de la réponse. Le deuxième est le DefaultHandler qui est une implémentation par défaut. Cette implémentation permet de décoder une suite d'octets formant un objet (ou structure en Rust) au format JSON et d'envoyer cette structure décodée sur un canal d'événements pour être traitée par une autre entité. Il attend ensuite, via un autre canal, la réponse de cette entité afin de pouvoir la renvoyer. Il est aussi possible de créer son propre Handler afin d'avoir le contrôle total sur les octets arrivant sur une connexion.

Ce deuxième Handler constitue une pièce maîtresse dans l'élaboration du système de fonctionnement événementiel répartit à travers cette nouvelle application Kollaps.

4.4 Fonctionnement événementiel

Les différents composants de Kollaps fonctionnent généralement par un système événementiel. Le principe est qu'ils écoutent sur un canal d'événements qui est alimenté soit par eux-mêmes, soit par un élément externe. Cet élément externe peut être un autre composant ou un Handler.

Cette technique permet de centraliser la gestion des événements dans une seule coroutine. Effectivement, l'application Kollaps est composée de plusieurs coroutines qui peuvent toutes produire des événements. La centralisation de la gestion des événements fait office de système de synchronisation pour accéder à un état partagé.

Chaque composant défini son propre type d'événements (sa propre structure).

Typiquement, un des producteurs d'événements le plus répandu est le Handler, expliqué au chapitre précédent (ch. 4.3). Un composant connecté au réseau, comme le CManager par exemple, écoute les connexions entrantes sur son port dédié. À chaque nouvelle connexion, un Handler y est associé, le DefaultHandler. Ce DefaultHandler décode les données transmises pour ensuite les insérer dans le canal d'événements du CManager. C'est ensuite au CManager de s'occuper de cet événement. Il se peut que plusieurs connexions simultanées apparaissent et donc, plusieurs événements sont produits en même temps. Avec cette technique, tous les événements sont traités séquentiellement et cela permet de définir une causalité des événements. Nous passons d'un environnement concurrent à un environnement séquentiel afin de garantir l'intégrité des données.

Ce système peut avoir des effets sur la performance, si nous avons énormément d'événements concurrents ou si chacun des événements demande un grand temps de calcul.

Ce n'est pas notre cas. Les événements sont peu fréquents et si certains ont une longue durée de traitement, il est possible de les exécuter dans d'autres thread et de continuer à s'occuper des événements qui n'ont pas de conséquence sur le calcul en cours. De plus, l'intégrité des données est primordiale dans notre cas, car l'état du cluster est enregistré à un endroit unique, sur le leader.

La Figure 19 ci-dessous montre la mise en place du système événementiel pour un composant quelconque, nommé *Component*. Ce composant met d'abord en place l'écoute des connexions entrantes par rapport à un protocole et définit le DefaultHandler comme producteur d'événements sur le canal *event_channel*.

Il faut noter que les deux boucles (bleue et jaune) sont des exécutions concurrentes.

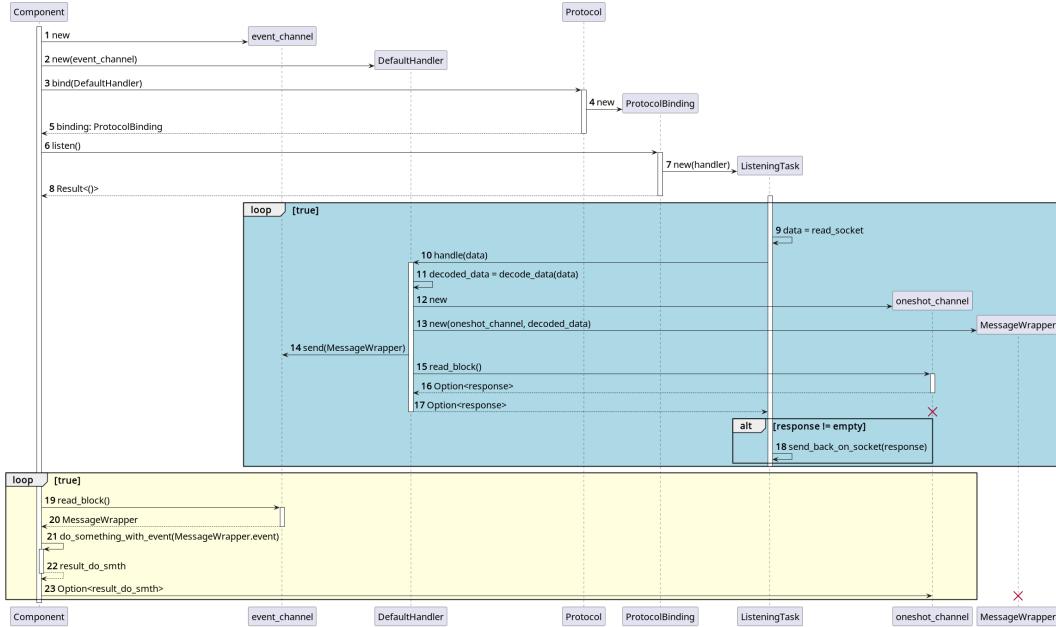


Figure 19: Mise en place d'un système événementiel. Agrandissement disponible en annexe, ch. 11.3.1 Figure 51

Ce système événementiel est utilisé par l'OManager, pour gérer séquentiellement les nouvelles expérimentations à orchestrer, essentiel pour maintenir un graphe d'expérimentations valide. Il est aussi utilisé par l'EManager pour protéger la liste des émulations en cours aux accès concurrents. L'EmulCore utilise ce système afin de maintenir le graphe d'utilisation de la bande passante cohérent. Et finalement, le CManager utilise cette technique pour accepter séquentiellement des nouvelles machines dans le cluster Kollaps.

4.5 CManager - Gestion du cluster

Le but du Cluster Manager est de maintenir l'état du cluster Kollaps. Il permet à la machine leader de recevoir les demandes d'adhésion dans le cluster et de vérifier que les machines présentes sont toujours en état de marche. Pour les machines non-leader, il permet de faire des demandes d'adhésion, d'effectuer des tests de performance avec d'autres machines et de répondre aux requêtes du leader.

Le CManager est la pièce centrale de la gestion du cluster. C'est cet élément qui réceptionne et traite les événements liés à la gestion du cluster. Il a cependant besoin de deux autres blocs : le contrôleur des tests de performance et le CGraph.

Le CGraph ou *Cluster Graph* est une implémentation de graphe de bande passante utilisée pour maintenir la connaissance des états de connexion entre les machines du cluster. Cette structure implémente l'algorithme (Alg. 2) défini dans le chapitre 3.7.2. Il est possible d'indiquer au graphe la bande passante obtenue après un test entre deux nœuds du cluster et il va ensuite être capable de mettre à jour la bande passante entre les différents nœuds du graphe en fonction de cette nouvelle donnée (extrapolation).

Le deuxième bloc est le PerfCtrl. Le PerfCtrl est appelé par le CManager pour exécuter un test de performance entre lui et une autre machine. Le contrôleur se charge de déployer, gérer et utiliser une instance du logiciel iPerf3. Le contrôleur peut soit fonctionner en mode serveur, soit en mode client.

Une fois un test exécuté, que ce soit en mode client ou serveur, le contrôleur détruit

l'instance d'iPerf3 afin d'économiser de l'utilisation CPU. Le PerfCtrl récupère ensuite la sortie de l'exécution d'iPerf3 pour trouver le résultat du test.

Le PerfCtrl a accès directement au Network Helper afin de pouvoir directement communiquer avec la machine avec laquelle il doit faire le test.

La Figure 20 ci-dessous montre le diagramme de séquence exécuté par le contrôleur de performance lorsque le CManager lui demande d'effectuer un test de performance.

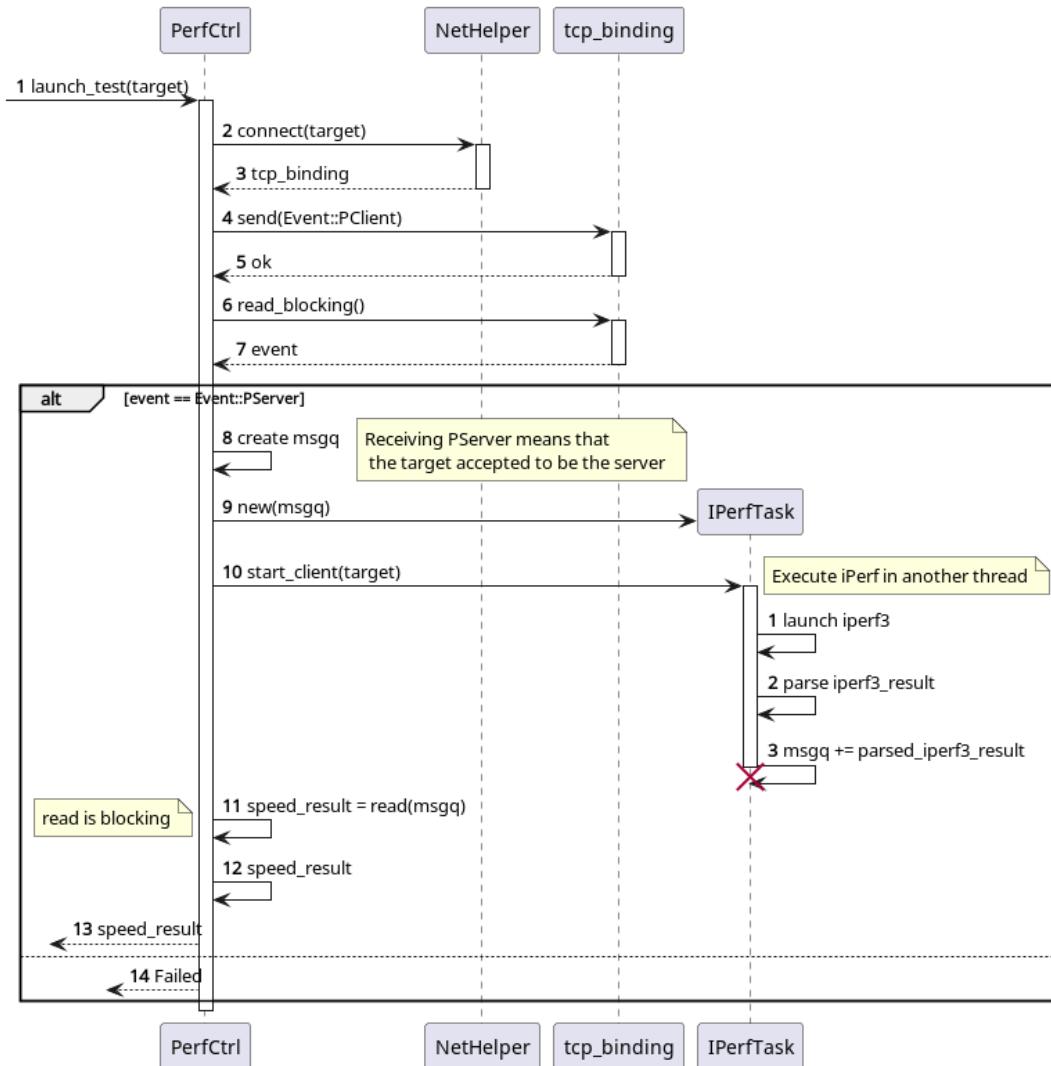


Figure 20: Diagramme de séquence montrant le lancement d'un test en mode client par le PerfCtrl

Comme expliqué plus haut, le CManager fonctionne avec un système événementiel. Les événements produits par le CManager lui-même sont appelés des événements à usage ou utilisation local(e). Ces différents événements sont décrits dans le chapitre suivant.

4.5.1 Événements du CManager

Le CManager définit plusieurs événements qui peuvent être classés en quatre catégories :

- Événements liés au CGraph

- CGraphUpdate(cgraph) : mise à jour du CGraph, utilisé par nouveau nœud du cluster pour signaler au leader une mise à jour du graphe après avoir effectué des tests de performance.
- CGraphGet : utilisé par un nouveau nœud du cluster pour demander au leader de lui fournir le CGraph actuel afin de pouvoir commencer les tests de performance.
- CGraphSend(cgraph) : réponse du leader à une requête CGraphGet.
- Événements liés à la demande d'entrée dans le cluster (CJQ : Cluster Joining Query)
 - CJQRequest(ClusterNodeInfo) : demande d'entrée dans le cluster, broadcastée par un nouveau nœud en indiquant son adresse au format ClusterNodeInfo.
 - CJQResponse(CJQResponseKind) : réponse du leader à une requête d'adhésion, le type (CJQResponseKind) peut être soit *Accepted* pour autoriser la procédure ou *Wait* pour demander de réessayer plus tard. *CJQRetry* : événement à utilisation locale créé et inséré dans le canal d'événements par un nouveau nœud qui a soit reçu un *Wait* du leader, soit qui a échoué sa dernière tentative de connexion.
 - CJQAbort : envoyé par un nouveau nœud au leader lorsqu'il détecte un dysfonctionnement de son côté lors de la procédure d'adhésion dans le cluster.
- Événements liés au maintien de l'état du cluster
 - CHeartbeat(ClusterNodeInfo, Uuid) : événement broadcasté par le leader en y insérant son adresse sous forme de ClusterNodeInfo et l'id du cluster. Suite à la réception de cet événement, un nœud du cluster renvoie immédiatement le même événement avec ses informations.
 - CHeartbeatCheck : événement à utilisation locale créé et inséré dans le canal d'événements par le leader. Cet événement est périodiquement inséré dans le canal et signifie qu'il est temps d'effectuer un test heartbeat.
 - CHeartbeatReset : événement à utilisation locale créé par le leader signalant que le compteur de CHeartbeat manqués par un nœud du cluster doit être remis à zéro.
 - NodeFailure : événement à utilisation locale créé par le leader signalant qu'un nœud du cluster n'a pas répondu à une pulsation Heartbeat.
- Événements liés aux tests de performance
 - PClient : demande de test envoyée à un autre nœud du cluster. Signifie que l'émetteur de l'événement voudrait exécuter un test et y participer en tant que client iPerf3.
 - PServer : acceptation d'une demande de test de performance. Signifie que l'émetteur de l'événement accepte de participer à un test de performance en tant que serveur.

Le CManager s'occupe de garder l'état du cluster et de détecter les changements. C'est lui qui maintient (sur le leader) la liste des nœuds Kollaps formant le cluster. Chaque changement peut avoir des implications pour les autres composants de Kollaps. C'est pourquoi, le CManager peut recevoir en paramètre un canal par le biais duquel il va indiquer les changements du cluster lors de leur détection. Pour ce faire, il insère sur ce canal un certain type d'événement :

- CGraphUpdate::New(cgraph) : permet d'indiquer qu'un nouveau nœud a rejoint le cluster. Le nouveau graphe est transmis en entier.

- CGraphUpdate::Remove(ClusterNodeInfo) : permet d'indiquer qu'un nœud n'est plus accessible ou qu'il a quitté le cluster.

Ces événements permettent de coordonner les CManagers répartis sur les différents nœuds du cluster Kollaps lors des différentes procédures. Ces procédures sont : l'adhésion au cluster et la détection de nœuds en échec.

4.5.2 Adhésion

Lorsqu'une instance de Kollaps est démarrée en mode non-leader, le CManager cherche à se connecter à un cluster Kollaps existant dans son réseau. L'explication ci-dessous se réfère au diagramme de séquence présent sur la Figure 21.

Pour ce faire, la découverte d'un cluster s'effectue en utilisant le protocole UDP et en broadcastant une requête d'adhésion (CJQRequest). Uniquement le leader d'un potentiel cluster Kollaps existant répond à ce type de requêtes.

Comme le protocole UDP n'est pas fiable, le nouveau nœud réessaie d'envoyer une requête d'adhésion si la précédente n'a pas eu de réponse. Le nombre d'essais est configurable.

Le nouveau nœud peut aussi recevoir une réponse de type *Wait* du leader. Cette réponse signifie que le nouveau nœud doit attendre avant de réessayer, car il y a déjà une procédure d'adhésion en cours par un autre nœud.

La partie découverte du cluster et réponse du leader se déroule avec le protocole UDP.

Une fois que le nouveau nœud reçoit une réponse positive à sa demande, il cherche à récupérer l'état actuel du cluster (CGraph) auprès du leader.

Afin de garantir une transmission correcte des données entre le leader et un nouveau nœud, la requête de l'état du cluster (CGraphGet) et sa réponse (CGraphSend) se déroulent en utilisant le protocole TCP. La suite des échanges utilise aussi ce protocole.

Une fois que le CGraph est récupéré, le nouveau nœud commence à exécuter des tests de performance avec les autres nœuds du cluster en utilisant l'algorithme 1 décrit au chapitre 3.7.2, jusqu'à compléter le CGraph. Cette nouvelle version du CGraph est ensuite renvoyée au leader via l'événement CGraphUpdate. Cet événement définit la fin de la procédure d'adhésion du côté du nouveau nœud lors de son envoi, et lors de sa réception du côté du leader.

Dans le cas où une erreur se produit dans quelque étape de cette procédure sur le nouveau nœud, celui-ci envoie un message d'annulation de procédure via l'événement CJQAbort.

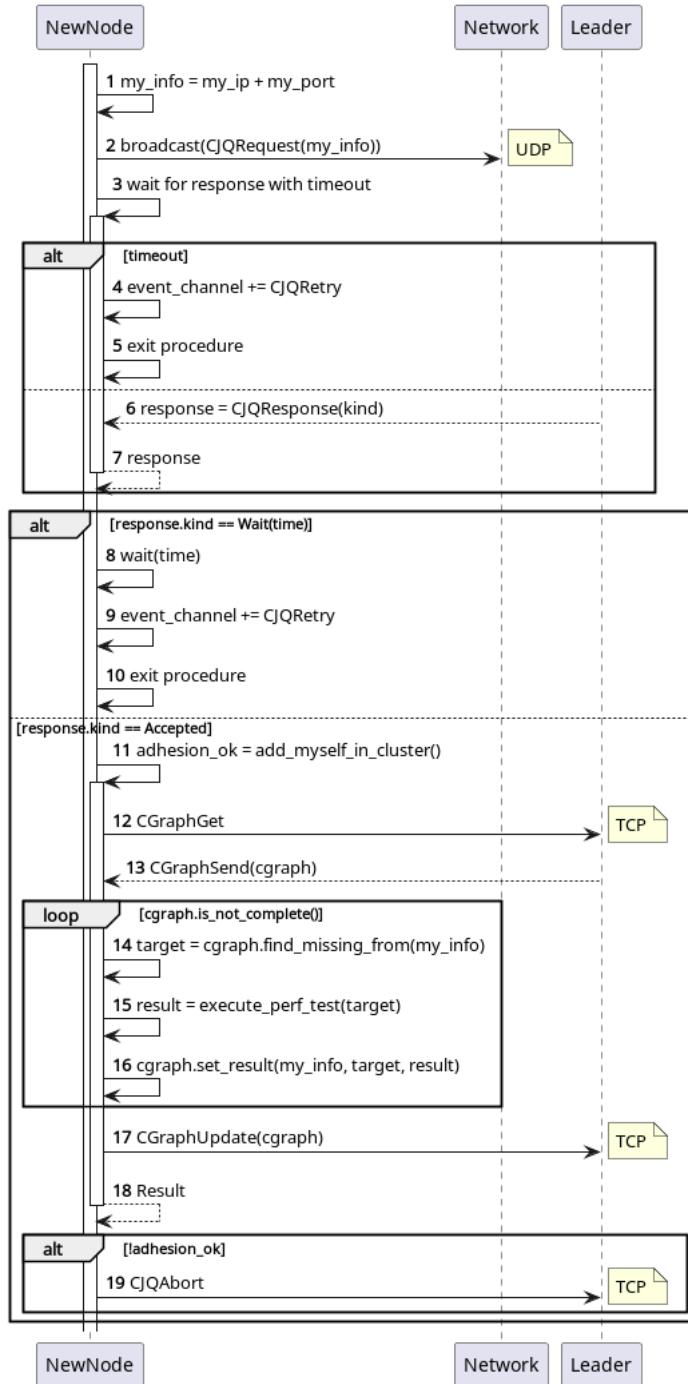


Figure 21: Procédure d'adhésion à un cluster Kollaps

En plus de la gestion des demandes d'adhésion, le CManager du leader doit aussi s'assurer que tous les nœuds du cluster sont toujours présents et fonctionnels.

4.5.3 Détection de nœuds en échec

Afin de détecter si certains nœuds sont en échec ou ne sont plus accessibles, le leader du cluster Kollaps utilise un système de battement de cœur (heartbeat). Il diffuse (broadcast)

à intervalles réguliers des événements CHeartbeat sur le réseau.

Ces événements transportent avec eux l'identifiant du cluster auquel ils font référence, cela permet de gérer plusieurs clusters sur un même réseau. Lorsqu'un CManager reçoit cet événement, il répond aussi tôt avec le même événement à l'émetteur en y insérant ses données.

Après l'émission d'un événement CHeartbeat, le leader attend une durée déterminée (configurable), avant de vérifier tous les messages reçus en retour. Il ne lui reste qu'à passer à travers les réponses et vérifier que tous les nœuds ont bel et bien répondu.

Dans le cas où un nœud Kollaps n'a pas répondu, un événement NodeFailure est inséré dans le canal des événements pour être traité par la boucle principale du CManager. Le CManager incrémente, à la réception de cet événement, le compteur du nombre de réponses manquées pour le nœud en question. Si un nœud dépasse le nombre de réponses manquées autorisées (configurable), il est considéré comme en échec. Il est essentiel d'autoriser les nœuds à manquer une ou plusieurs réponses, car la communication se fait en UDP et les paquets ne sont pas garantis d'arriver à destination ou d'arriver avec des données correctes[14].

Lorsqu'un nœud ayant manqué un événement CHeartbeat répond à nouveau, le CManager du leader remet à zéro son compteur avec l'événement CHeartbeatReset.

L'événement CHeartbeatCheck est l'événement qui déclenche le contrôle du cluster. À la fin de la procédure de contrôle, cette même procédure insère à nouveau l'événement dans le canal des événements. C'est ce qui produit la répétition du contrôle à intervalles réguliers.

Lorsque nous savons de quoi est constitué le cluster Kollaps grâce au CManager, il est possible de commencer à répartir des émulations sur les différents nœuds en fonction de leurs connexions. C'est le rôle du OManager.

4.6 OManager - Gestion de l'orchestration

Le rôle de l'OManager est d'accepter de nouvelles expérimentations et de trouver une façon de les déployer sur le cluster Kollaps. Il s'agit de l'unique composant qui doit offrir un point d'entrée fixe pour une communication avec des potentiels clients qui veulent soumettre de nouvelles expérimentations.

Afin de pouvoir trouver une répartition possible d'une expérimentation Kollaps sur le cluster, il fait appel à un orchestrateur. La Figure 22 ci-dessous montre le diagramme de classe du composant d'orchestration. Les éléments importants ne sont pas chacune des méthodes ou des attributs, mais le fait que l'agrégation entre l'OManager et l'orchestrateur a pour cardinalité "1 - 1". Cela signifie qu'il aurait été possible d'intégrer toutes les fonctionnalités de l'orchestrateur à l'OManager.

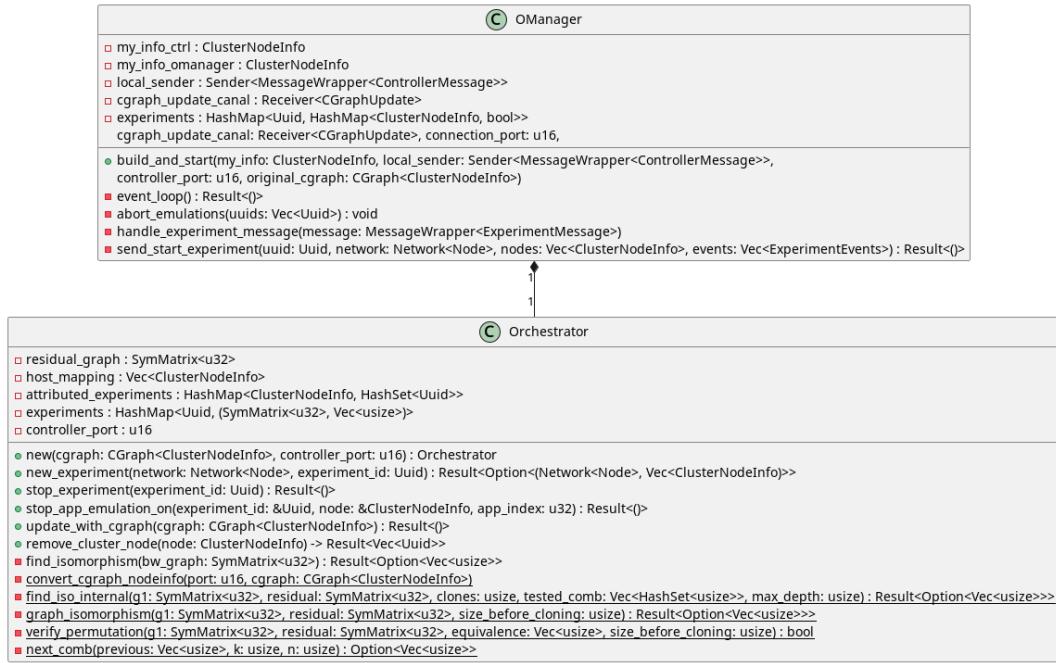


Figure 22: Diagramme de classe du composant d'orchestration

Cette découpe est faite de sorte que l'orchestreur fonctionne en mode boîte noire. Cette séparation des tâches rend l'implémentation plus élégante en ayant une partie s'occupant de la gestion des événements (OManager) et une autre partie s'occupant uniquement de l'orchestration.

L'orchestrateur actuel utilise la technique d'orchestration décrite dans le chapitre 3.8. Avec cette conception, il est possible dans l'avenir de complètement changer d'implémentation sans devoir modifier la logique de gestion des événements.

4.6.1 Événements de l'OManager

L'OManager supporte deux types d'événement. Le premier, les événements CGraphUpdate déjà décrits dans les événements du CManager. Ces événements indiquent les changements au niveau des machines formant le cluster. Il est essentiel pour l'OManager de recevoir des notifications de changements du cluster Kollaps, car chacun de ces changements implique soit que certaines expérimentations sont en échec dû au départ d'un nœud, soit le fait qu'il y a un nouveau nœud que l'orchestrateur peut utiliser pour répartir les expérimentations.

Le second type d'événements est celui lié aux expérimentations. Ces événements sont décrits ci-dessous :

- NewTopology : utilisé par un client externe, demande de déploiement d'une nouvelle expérimentation.
- Abort(experiment_uuid) : utilisé soit par un client externe demandant d'annuler une expérimentation, soit par un EManager qui indique que la partie d'expérimentation qu'il est en train d'émuler a subi un échec.
- CleanStop(experimentation_uuid, ClusterNodeInfo) : utilisé par un EManager pour indiquer que la partie d'une expérimentation qu'il doit émuler s'est terminée sans problème. Cet événement permet à l'orchestrateur de libérer des ressources.

- `EmulationReady(experiment_uuid, ClusterNodeInfo)` : utilisé par l'EManager pour indiquer que la partie de l'expérimentation qu'il doit émuler est prête.

En plus des tâches principales de l'OManager (orchestration et déploiement), celui-ci doit aussi assurer d'autres rôles qui sont fondamentaux au bon fonctionnement d'une expérimentation.

4.6.2 Rôles

Comme décrit dans les chapitres précédents, les rôles principaux de l'OManager sont l'orchestration et le déploiement de nouvelles expérimentations.

Ces rôles s'accompagnent de "sous-rôles" qui permettent le bon fonctionnement d'une expérimentation.

L'OManager agit aussi comme la partie centralisée de cette nouvelle version de Kollaps (Figure 17). Le premier sous-rôle est la coordination des parties d'expérimentation.

Quand l'OManager répartit des parties d'une expérimentation sur les différentes machines du cluster, il sert de barrière de synchronisation pour ces parties. Effectivement, les tailles (sous-entendu le nombre de processus à gérer) de ces parties peuvent être déséquilibrées, ce qui implique que le temps de mise en place d'une partie d'expérimentation peut varier en fonction du nœud et de la partie.

Lorsqu'un EManager a fini d'initialiser une partie d'expérimentation, il l'indique à l'OManager. L'OManager peut ainsi attendre que chacune des parties d'une expérimentation soit prête pour autoriser le départ de l'expérimentation.

Le second sous-rôle est la gestion des erreurs. Si une partie d'expérimentation tombe en échec sur l'un des nœuds, ce nœud l'indique à l'OManager. C'est ensuite le rôle de l'OManager de signaler l'échec et l'annulation de l'expérimentation à tous les nœuds concernés. De cette manière, il peut aussi indiquer à l'orchestrateur qu'une expérimentation est tombée en échec et celui-ci peut donc libérer les ressources qu'il avait allouées.

Nous pouvons voir la mise en place de chacun de ces rôles et sous-rôle sur le diagramme ci-dessous (Figure 23). Le diagramme montre la boucle principale de l'OManager qui écoute sur deux canaux d'événements en simultané : *l'exp_event_channel* pour les événements liés aux expérimentations, et le *cgraph_update_channel* pour recevoir les changements du cluster. La lecture concurrente sur ces deux canaux est indiquée par les couleurs bleue et jaune.

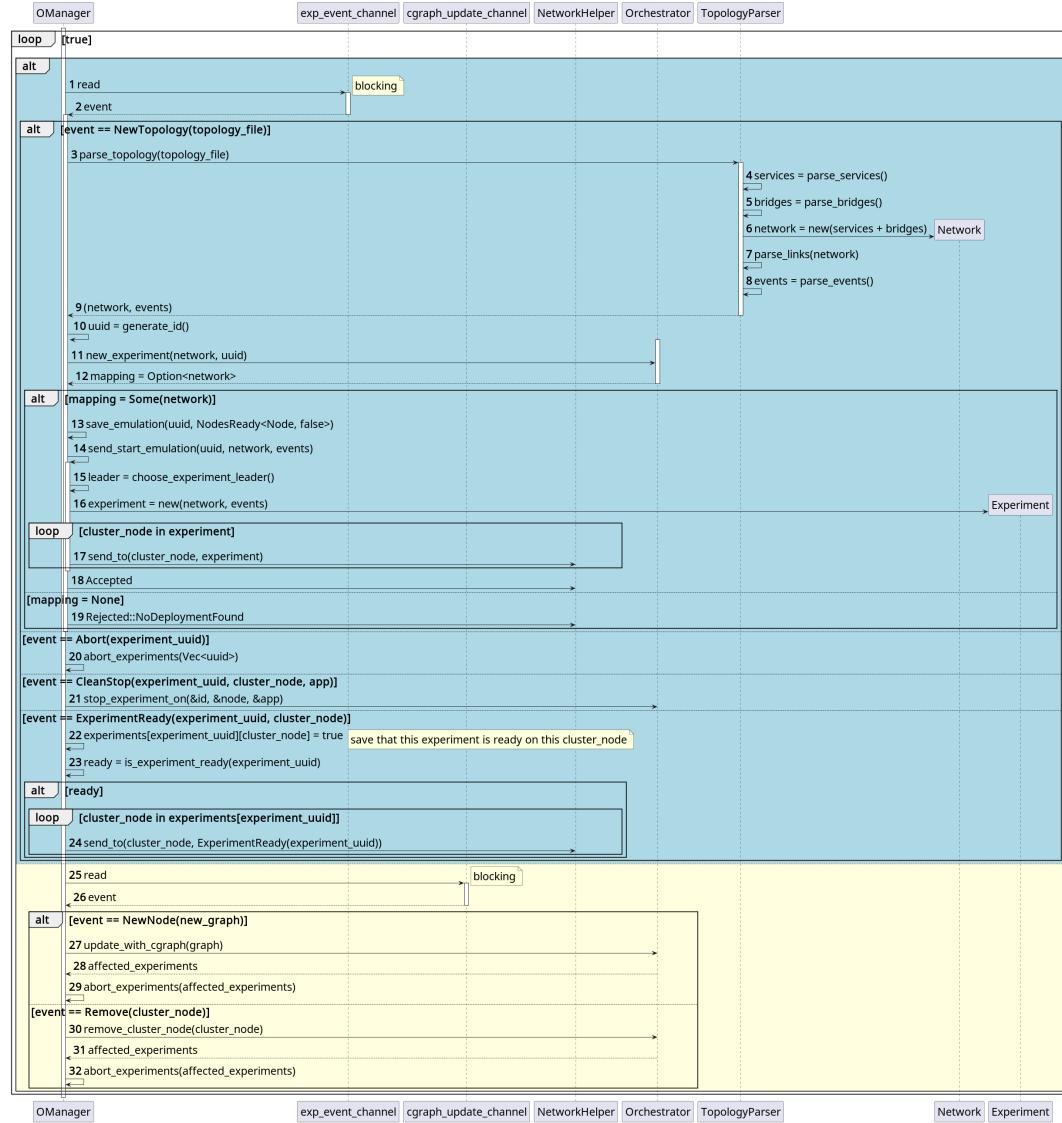


Figure 23: Boucle principale de l'OManager. Agrandissement en annexe, ch. 11.3.2 Figure 52

L'OManager est le premier composant en contact avec une nouvelle expérimentation. Après avoir trouvé une orchestration, il distribue les parties d'une expérimentation aux différents EManagers.

4.7 EManager - Gestion des émulations

Contrairement à l'OManager, l'EManager est présent sur chacun des nœuds du cluster Kollaps, y compris sur le leader.

L'EManager fait partie du composant de gestion des émulations. Son rôle est de recevoir les événements envoyés par l'OManager et d'agir en fonction. L'EManager ne s'occupe pas directement de l'émission, mais sert de support pour les EmulCores.

Pour chaque partie d'expérimentation que l'EManager reçoit de l'OManager, il crée un EmulCore qui s'occupe d'émuler cette partie d'expérimentation. Cette conception permet

aux noeuds de supporter plusieurs parties provenant chacune d'une expérimentation différente et donc, permet au cluster Kollaps de supporter plusieurs émulations simultanées.

Les rôles principaux de l'EManager sont de maintenir une liste d'EmulCore en cours de fonctionnement, de détecter l'échec d'un EmulCore, de démarrer de nouveaux EmulCores et vérifier leur état d'initialisation et d'intercepter et rediriger les messages destinés aux EmulCores. Il doit aussi se charger, en cas d'annulation d'une expérimentation, d'arrêter correctement un EmulCore.

Pour exécuter ces tâches, il se base lui aussi sur un système événementiel comme présenté dans le chapitre 4.4.

4.7.1 Communication et événements de l'EManager

L'EManager initie des communications réseau uniquement en direction de l'OManager présent sur le leader. Cependant, il peut recevoir des messages provenant d'EmulCores présents sur d'autres machines. Jamais un EManager ouvrira un canal de communication en direction d'un autre EManager dans le cluster.

Les EmulCores dispersés sur les noeuds Kollaps et collaborant pour une même expérimentation s'échangent constamment des messages pour indiquer leur utilisation de bande passante. Au lieu d'ouvrir un canal de communication réseau directement entre eux, ils envoient leurs messages aux EManagers qui, eux, redirige le message vers le bon EmulCore. Ce fonctionnement a deux avantages :

- Pas besoin d'allouer un port pour chaque expérimentation, ce qui diminue le nombre de ports utilisés et réduit le risque de conflits.
- Si un EmulCore s'est arrêté correctement sur une machine, le message est tout simplement ignoré par l'EManager. Dans le cas où un EmulCore n'est pas accessible et n'est pas censé s'être arrêté, l'EManager peut détecter sa mise en échec.

Pour ce faire, avant d'envoyer un message, un EmulCore englobe (wrap) le message dans un événement appelé EmulCoreInterchange qui contient le message original et l'identificateur de l'expérimentation afin de pouvoir être redirigé par l'EManager qui le reçoit.

La liste ci-dessous montre les événements utilisés par l'EManager.

- EmulCoreInterchange(experiment_uuid, EmulMessage) : envoyé par un EmulCore d'une autre machine en direction d'un EmulCore présent sur la machine de réception.
- ExperimentNew(Emulation) : envoyé par l'OManager pour demander de créer une nouvelle partie d'expérimentation.
- ExperimentStop(experiment_uuid) : envoyé par l'OManager pour demander d'arrêter une émulation en cours. Message généré lorsque l'OManager reçoit un Abort pour une expérimentation.
- ExperimentReady(experiment_uuid) : envoyé par l'OManager pour indiquer que chacune des parties d'une expérimentation sont prêtes. Cela signifie que l'EManager doit lancer l'EmulCore concerné par cette expérimentation.

L'EManager est l'élément chargé de gérer une flotte d'EmulCores, qui, eux, se chargent d'exécuter une émulation Kollaps.

4.8 EmulCore - Gestion locale d'une émulation

Un EmulCore gère une partie d'expérimentation Kollaps. Il est démarré par l'EManager mais est capable de gérer son propre cycle de vie après son lancement.

Le cycle de vie d'un EmulCore est constitué de trois parties (voir Figure 24).

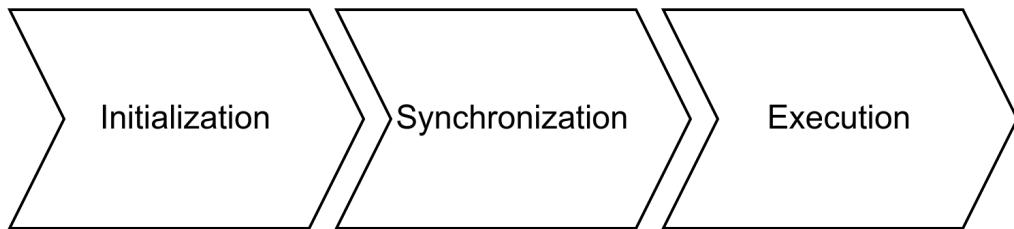


Figure 24: Cycle de vie d'un EmulCore

Lors de l'initialisation, l'EmulCore analyse la structure représentant une émulation pour en extraire différentes données, telles que les PTs qu'il doit déployer / surveiller, les autres nœuds qui hébergent aussi des parties de cette même expérimentation, etc.

Après avoir extrait ces données, l'EmulCore vérifie la présence des PTs, et si nécessaire, se charge de les lancer.

Ensuite, une fois que chaque PT est prêt, il lance l'exécution des *Reporters*, un par PT. Le but des Reporters est d'analyser l'utilisation réseau du processus auquel il est lié, et de retransmettre ses analyses à l'EmulCore.

Une fois que tout est en place, l'EmulCore enregistre un canal de communication auprès de l'EManager afin que celui-ci puisse lui retransmettre les messages en provenance d'autres EmulCores.

L'EmulCore est considéré comme initialisé et l'EManager peut le signaler à l'OManager.

Une fois que l'OManager envoie la commande de démarrage d'une expérimentation, l'EManager prend l'EmulCore et déclenche sa partie synchronisation. Cette partie a pour but de permettre aux EmulCores participant à une même expérimentation de se mettre d'accord sur l'heure de commencement de l'émulation afin qu'ils soient synchronisés.

La synchronisation suit l'algorithme décentralisé décrit au chapitre 3.10. À la fin de cette phase, l'EmulCore est considéré comme synchronisé.

Une fois synchronisé, il passe directement dans sa phase d'exécution. Il commence par instancier un planificateur qui émettra les événements de l'expérimentation en temps voulu. Une fois le planificateur prêt, il se met à écouter sur les événements produits par les différents Reporters ou les autres EmulCores.

Finalement, une émulation est détectée comme finie une fois que tous les PTs qu'un EmulCore doit surveiller se mettent en mode *crash* via les événements de l'expérimentation. Le mode *crash* ne signifie pas forcément qu'un processus terminal est en échec, mais qu'il quitte l'expérimentation. Dans le cas où le cycle de vie du processus est géré par l'EmulCore, il est arrêté à ce moment-là.

C'est aussi à ce moment-là que l'EmulCore demande au Reporter d'annuler la configuration du linux traffic control et de s'arrêter. Un message CleanStop est finalement envoyé à l'OManager pour que l'orchestrateur puisse libérer les ressources allouées.

4.8.1 Communication avec un Reporter

Le Reporter est une application déployée dans l'espace de nom du PT auquel il est lié. Le transfert d'information entre lui et l'EmulCore se fait via des sockets Unix. Les sockets Unix permettent de transférer des messages sans passer par le réseau, mais directement

via la mémoire vive [9]. La librairie NetHelper implémente l'utilisation des sockets Unix, ce qui permet aux deux éléments d'utiliser la même librairie pour communiquer.

Lors de son démarrage, l'EmulCore crée un socket Unix appelé *flow_socket*. Ce socket est fourni aux différents Reporters et leur permet d'y insérer des événements lorsque leurs analyses indiquent des changements dans l'utilisation du réseau.

Aussi, durant son initialisation, le Reporter crée un socket Unix. Ce socket est ensuite enregistré par l'EmulCore comme moyen de communication avec un Reporter en particulier. Le socket est utilisé par l'EmulCore afin de transmettre au Reporter des ordres concernant la modification des valeurs du linux traffic control.

Ces ordres sont envoyés en fonctions des changements aperçus dans l'utilisation de la bande passante. Le calcul de cette utilisation est facilité par la librairie *Bandwidth Sync Helper*.

Un EmulCore réagit, comme la plupart des composants, à certains événements. La différence avec les autres composants est que ces événements peuvent provenir de plusieurs sources.

4.8.2 Exécution et événements

Lors de son exécution, un EmulCore réagit séquentiellement aux événements qui lui parviennent. Ces événements sont au nombre de deux.

- FlowUpdate(source, destination, bande passante) : provient soit d'un autre EmulCore, soit d'un Reporter. Dans les deux cas, cela indique qu'il y a eu changement d'utilisation de bande passante entre la source et la destination. La bande passante indiquée dans l'événement est la bande passante désirée par le flux. L'EmulCore peut ensuite faire ses calculs pour mettre à jour son état de l'émulation. Lorsqu'il provient d'un Reporter, la mise à jour est broadcastée aux autres nœuds et le Reporter reçoit des ajustements du linux traffic control si nécessaire.
- Event(ExperimentEvent) : événement produit par le planificateur qui représente un événement de l'expérimentation. Il transporte avec lui la nature de l'événement, à savoir *Join*, *Quit* ou *Crash*.

L'EmulCore se charge de faire les calculs d'utilisation de la bande passante et s'occupe de définir les valeurs que le linux traffic control doit prendre pour chaque processus terminal. Néanmoins, il est incapable de modifier les valeurs du linux traffic control, car il ne se trouve pas dans le même espace de nom que le PT en question. C'est pourquoi il fait appel à un Reporter.

4.9 Reporter et Monitor - Récupération des données réseau

Le Reporter ne peut pas fonctionner sans le Monitor et le Monitor n'a pas raison d'exister sans Reporter. Il s'agit d'un combo d'applications qui travaillent ensemble. Le Monitor est une application eBPF. Cette application est déployée par le Reporter lors de son lancement. Le Monitor se charge d'analyser les paquets réseau et de procéder à de simples calculs afin de déterminer la consommation instantanée de bande passante.

À intervalles réguliers (toutes les 25 millisecondes), le Monitor informe le Reporter de l'utilisation réseau. Le Reporter peut ainsi détecter les changements dans l'utilisation de bande passante. Comme dans la précédente version de Kollaps, ces changements sont signalés uniquement lorsqu'ils montrent une variation suffisante par rapport l'utilisation actuelle, à la différence que dans cette version, cette variation est configurable.

Comme un Reporter est déployé dans le même espace de nom réseau que le processus qu'il doit surveiller, c'est à lui que revient la tâche de moduler les valeurs du linux traffic

control.

Un Reporter écoute sur plusieurs fronts en même temps. Tout d'abord, il écoute sur les informations en provenance du Monitor et concurremment, il écoute sur les ordres de l'EmulCore.

L'EmulCore peut émettre cinq ordres différents :

- TCInit(TCConf) : initialisation du linux traffic control pour une destination.
- TCUpdate(TCConf) : mise à jour des paramètres pour une destination.
- TCDisconnect : émulation de la déconnexion de l'espace de nom en simulant un lien cassé.
- TCReconnect : remise en état de la configuration du TC après un TCDisconnect.
- TCTeardown : indique de supprimer les modifications apportées au TC et d'arrêter le Reporter.

La structure *TCConf* utilisée par l'EmulCore pour exprimer une configuration du TC est représentée sur la Figure 25 ci-dessous.

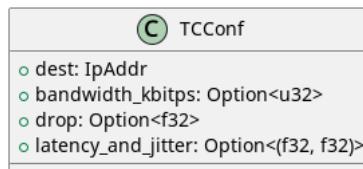


Figure 25: Représentation d'une configuration TC

4.9.1 Détection des changements d'utilisation de bande passante

Comme exprimé dans le chapitre précédent, le Monitor rapporte l'utilisation **instantanée** de bande passante toutes les 25 millisecondes. La nouvelle version du Monitor classe directement l'utilisation de bande passante en fonction de la destination.

Le Reporter maintient un HashMap d'usage. Les clés du HashMap sont les destinations, et les valeurs, la consommation actuelle de bande passante associée à un estampillage indiquant la date de la dernière mise à jour.

À chaque notification du Monitor, cet HashMap d'usage est mis à jour.

Une autre tâche tourne concurremment et consiste à traverser ce HashMap d'usage afin d'y détecter des changements. Cette tâche maintient un autre HashMap qui contient les valeurs du HashMap d'usage lors de la dernière traversée. Cela permet de comparer les nouvelles valeurs avec les précédentes.

Le Reporter détecte ainsi la fin d'un flux vers une destination lorsque l'utilisation de bande passante n'a pas été mise à jour depuis plus d'un certain temps (configurable). Une fois la fin d'un flux détecté, il est supprimé de l'HashMap d'usage.

Il détecte un nouveau flux lorsqu'une entrée apparaît dans l'HashMap d'usage, mais pas dans l'HashMap qui contient les anciennes valeurs.

Finalement, il détecte une variation lorsque les valeurs d'utilisation sont différentes entre les deux HashMap pour une même destination.

Cette technique permet de facilement détecter de nouvelles entrées ou des arrêts de flux, ainsi que des variations. Toutefois, elle implique des accès en écriture et lecture concurren-

rents sur une même structure de données. L'implémentation doit utiliser une structure efficace, si possible *lock-free*.

Le combo Reporter et Monitor est déployé dans l'espace de nom du processus qu'ils doivent surveiller. Pour ce faire, l'EmulCore doit garantir que ce processus est déjà démarré avant le lancement du combo. Certains PTs sont des conteneurs qui doivent être instanciés et arrêtés par l'EmulCore. Pour ce faire, il utilise une librairie nommée Dock-Helper.

4.10 DockHelper - Librairie d'aide aux communications avec le Docker Engine

La librairie DockHelper permet de faire le lien avec le Docker Engine et l'application. Pour ce faire, la librairie fournie une interface Rust et exécute des commandes Docker en arrière-plan. C'est un moyen simple de communiquer avec le Docker Engine.

C'est le rôle de l'EManager de créer une instance de cette librairie. Lors de sa création, DockHelper configure Docker pour créer un réseau de type IpVlan, nécessaire au bon fonctionnement de Kollaps.

4.11 Configuration

La plupart des composants ont besoin d'informations fournies par l'utilisateur. Typiquement, le DockHelper a besoin de connaître le sous-réseau dans lequel il doit créer un réseau IpVlan. D'autres éléments ont besoin de connaître leur port principal de communication. C'est pourquoi cette nouvelle application Kollaps demande à l'utilisateur d'indiquer certaines informations obligatoires, mais lui permet aussi de configurer des paramètres liés au fonctionnement général de l'application.

Afin de pouvoir facilement traiter les informations que l'utilisateur fournit, le programme utilise la librairie [Clap²¹](#) qui est aujourd'hui le standard pour gérer des configurations utilisateur en Rust. La librairie permet de facilement traiter des fichiers de configuration ainsi que des informations en ligne de commande.

La conception définit l'architecture de la nouvelle version de Kollaps et la communication entre ses composants. Elle définit les tâches de chacun et leur principe de fonctionnement. Le chapitre suivant (ch. 5) traite de l'implémentation de ces composants et de leur fonctionnement interne.

²¹<https://crates.io/crates/clap>

5 Implémentation

L'implémentation suit ce qui est présenté au chapitre précédent concernant la conception (ch. 4).

Ce chapitre décrit uniquement certains aspects de l'implémentation du logiciel développé durant ce projet. Il montre les idées et les concepts en place et non le code directement. Pour une documentation plus détaillée sur le code, il faut se fier à la documentation présente dans les fichiers sources et dans le README présents dans le répertoire Git du logiciel (voir les ressources ch. 10).

Cette section reprend les composants décrits par la Figure 16, à savoir le NetHelper, le CManager et le PerfCtrl, l'OManager et l'orchestrateur, l'EManager et les EmulCore et finalement le Reporter et son Monitor. Le document explique l'implémentation d'un ou deux points clés pour chacun de ces éléments.

L'ordre de documentation est le même que pour la conception. Le premier élément est le NetHelper.

5.1 Encodage et décodage des événements par le NetHelper

Comme la conception l'a présenté, la plupart des composants communiquent entre eux par l'envoi d'événements. Le NetHelper permet l'envoi des événements via plusieurs protocoles de transport. Cela signifie qu'il n'est pas possible de se baser sur le protocole sous-jacent pour transmettre nos données. C'est pour cette raison que le NetHelper intègre un protocole dédié au transfert d'événements.

Ce protocole est simple et s'adapte sans problème à tous les protocoles de transport. Le principe est que chaque événement est lié à un code unique. Lors d'un transfert d'événement, l'événement est converti en code et les données qu'il transporte sont converties au format JSON. L'utilisation du JSON est facilité par la librairie Serde²². Ensuite, la taille des données à transmettre est calculée.

Finalement, les données sont envoyées sous le format décrit par la Figure 26 ci-dessous.

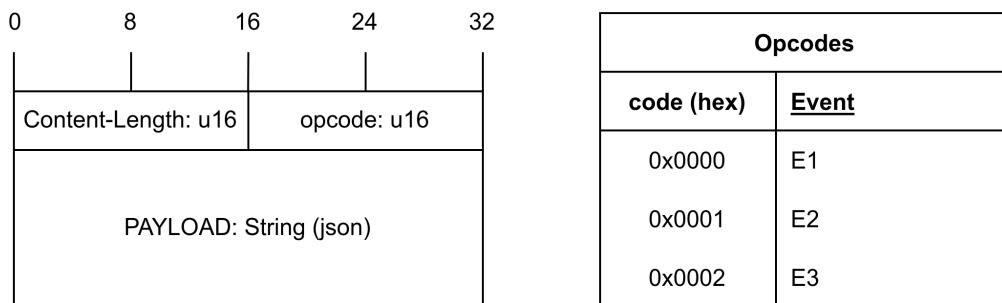


Figure 26: Constitution d'un événement envoyé sur le réseau

Le champ *content-length* est essentiel pour les protocoles fonctionnant sous forme de flux, comme le TCP. Il permet d'indiquer le nombre d'octets à lire afin de pouvoir constituer un paquet.

Lors de la réception, une fois qu'un envoie a été complètement lu grâce au champ *content-length*, l'*opcode* permet de recréer l'événement en y insérant ses données.

²²<https://serde.rs/>

Cet encodage est décodage est directement intégré au NetHelper et une implémentation par défaut pour les événements est disponible via l'interface *ToBytesSerialize*.

5.2 Gestion des Timeouts dans le CManager

Lors du démarrage d'un CManager (non-leader), celui-ci cherche à rejoindre un cluster existant dans son sous-réseau. Pour ce faire, il diffuse une requête à travers tout le réseau. L'envoi d'un broadcast se fait avec le protocole UDP qui ne garantit pas qu'un message soit livré. C'est pourquoi le CManager, après l'envoi de sa requête, attend une durée définie (configurable) pour recevoir une réponse. Si aucune réponse n'est reçue, alors il recommence.

L'environnement dans lequel fonctionnent les composants de Kollaps est asynchrone, au sens que l'entend Rust. C'est-à-dire que chaque partie de l'application fonctionne dans sa propre tâche et qu'une tâche peut être stoppée ou redémarrée par un planificateur (scheduler) qui se charge de répartir les tâches sur différents threads.

Cela permet d'effectuer plusieurs tâches concurremment.

La gestion des timeouts, se fait en utilisant cette propriété. Avant l'envoi de la requête, le CManager crée deux tâches (des *Futures*). La première envoie la requête sur le réseau et attend pour une réponse. Une fois qu'une réponse parvient, la tâche retourne la réponse et termine. La deuxième tâche ne fait que dormir la durée du timeout et se termine une fois qu'elle a fini de dormir. Les deux tâches sont *Cancellable*, ce qui signifie qu'elles peuvent être annulées en tout temps.

Une fois les deux tâches créées, il suffit de les lancer en même temps et attendre sur la première qui se termine. Pour ce faire, nous pouvons utiliser la primitive *select!* fournie par la librairie Tokio utilisée dans le projet.

5.3 PerfCtrl et le lancement d'iPerf3

Le PerfCtrl utilise iPerf3 pour effectuer des tests de performance. Cela signifie qu'il doit être capable de lancer une instance d'iPerf3 et de la supprimer une fois un test terminé. La partie intéressante se trouve du côté client du test.

La librairie NetHelper permet de définir un *Handler* pour une connexion. Ce Handler est exécuté lorsque des données arrivent sur la connexion. Normalement, ce Handler doit pouvoir être cloné, car si nous avons une connexion TCP, chaque flux est exécuté dans sa propre tâche et le Handler est cloné pour être transféré à chacune des tâches s'occupant d'un flux.

Cependant, le NetHelper autorise l'utilisation d'un Handler spécial, appelé un *ResponderOnce*. Un *ResponderOnce* est un type de Handler qui peut être exécuté qu'une seule fois et donc ne peut pas être cloné. Il s'agit en fait d'une *Closure* Rust de type *FnOnce*. Cela signifie que le Handler est capable d'encapsuler une partie de son environnement et de garder le contexte, mais avec la restriction de pouvoir être exécutée qu'une seule fois.

L'initiateur du test de performance peut alors créer un *ResponderOnce* qui, lors de son exécution, vérifie premièrement que l'événement reçu est de type PServer (acceptation de la demande de test). Ensuite, si c'est le cas, il lance iPerf3 en mode client. Une fois ce *ResponderOnce* créé, le PerfCtrl crée une connexion TCP en destination du récepteur du test de performance en passant le *ResponderOnce* comme Handler.

Il envoie un message PClient sur la connexion. Dès lors qu'une réponse est obtenue sur cette même connexion, le *ResponderOnce* va être automatiquement appelé et le test est garanti d'être synchronisé pour autant que la machine de destination démarre son serveur iPerf3 avant de répondre avec PServer. De cette manière, comme chaque flux dans une

connexion TCP est exécuté dans sa propre tâche, le test se déroule en arrière-plan sans interrompre le système.

Les avantages d'utiliser un ResponderOnce plutôt que de créer un nouvel objet Handler personnalisé sont :

1. La capture de l'environnement actuel. Moins de duplication, le ResponderOnce prend la propriété des données (passage de l'*ownership*).
2. La garantie que le test est exécuté qu'une seule fois.
3. Lisibilité du code.

La commande iPerf3 est exécuté grâce à la librairie standard de Rust qui nous permet de configurer une commande. Une fois une commande créée, Rust offre trois choix :

- Spawn
- Output
- Status

En fonction de notre choix, la commande sera exécutée différemment. Si nous choisissons *Spawn*, la commande est exécutée comme processus enfant et nous recevons un objet représentant ce processus. La méthode *Output* permet, elle, d'exécuter la commande aussi comme processus enfant, mais d'attendre sa terminaison tout en capturant son *StdOut* pour le retourner à la fin de l'exécution. Finalement, la méthode *Status* agit comme la méthode *Output*, sauf qu'au lieu de capturer la sortie standard, elle capture uniquement le code du status.

Dans le cas du PerfCtrl, c'est la méthode *Output* qui est utilisée. Ceci permet ensuite d'analyser le résultat du test de performance.

Pour ce qui est du côté serveur, le processus iPerf3 est démarré avec la méthode *Spawn*, car nous ne traitons pas le résultat. Aussi, afin de s'assurer que le serveur iPerf3 s'arrête après un test, le PerfCtrl démarre le processus iPerf3 avec l'option "-1" qui demande d'arrêter le serveur après un seul test.

5.4 Ne pas tester deux fois la même combinaison

Afin de simplifier l'implémentation de l'orchestrateur et la recherche d'une répartition d'une expérimentation, l'orchestrateur ne travail pas directement avec les identifiants des nœuds Kollaps. Lorsqu'il reçoit un CGraph, il lui demande de lui fournir une matrice des connexions réseau entre les nœuds Kollaps. Le CGraph fournit alors une matrice, ainsi qu'un vecteur servant de *mapper* vers les informations des nœuds. L'indice i dans la matrice représente le nœud se trouvant à l'indice i dans le vecteur *mapper*.

De cette manière, il est plus facile de simuler la création de clones. Nous n'avons pas besoin de modifier la matrice pour dupliquer certains nœuds, ce qui serait inefficace. Nous devons simplement prendre la taille du vecteur *mapper* et la multiplier par le nombre de clones nécessaires. Nous avons donc la fonction modulo qui permet de revenir au nœud original.

Prenons par exemple un cluster constitué de 3 nœuds et une expérimentation constituée de 3 processus terminaux. Dans le cas où la recherche d'isomorphisme n'a pas donné de résultat avec les trois nœuds originaux, nous procédons à une duplication de chacun des nœuds. Nous nous retrouvons alors avec un espace de recherche constitué de 6 nœuds.

Après la duplication, la recherche d'un isomorphisme recommence. L'algorithme va à nouveau essayer de trouver un isomorphisme en produisant toutes les combinaisons possibles de longueur 3 (pour trois PTs) sur un espace de recherche contenant 6 éléments.

Cela implique que l'orchestrateur essayera, sans le savoir, de trouver un isomorphisme en plaçant deux PTs sur un même nœud. Prenons l'exemple suivant où il trouve un isomorphisme en utilisant pour PT1, PT2 et PT3 le nœud 0, respectivement 1 et 3. Comme $0 \equiv 3 \pmod{3}$, alors les nœuds 0 et 3 sont en réalité un seul et même nœud en utilisant le vecteur *mapper*. Cela signifie que PT1 et PT3 sont déployés sur un même nœud.

L'utilisation des indices permet d'exprimer un nouvel espace de recherche à moindre coût et est beaucoup plus efficace que d'augmenter artificiellement la matrice des capacités réseau.

Du fait de l'augmentation artificielle de l'espace de recherche, il se peut que deux combinaisons soient équivalentes. En reprenant l'exemple cité ci-dessus, mais cette fois avec un espace de recherche de neuf éléments (un clonage supplémentaire), alors les combinaisons $\{0, 1, 3\}$ et $\{0, 1, 6\}$ sont équivalentes. Elles signifient placer deux PTs sur le nœud Kollaps 0 et un sur le nœud Kollaps 1.

Le but est de ne jamais tester deux fois la même combinaison. Le test d'une combinaison est très couteuse en temps de calcul, car il faut toujours générer toutes les permutations possibles au sein de la combinaison et les tester.

L'implémentation de la recherche d'isomorphisme de l'orchestrateur reçoit une combinaison en entrée et teste ensuite toutes les permutations de cette combinaison. Nous pouvons alors considérer les combinaisons testées comme des ensembles en les transformant. Plus concrètement, si la combinaison $\{0, 1, 3\}$ est passée à la recherche d'isomorphisme, nous pouvons considérer que les neuf permutations possibles sont testées. En transformant les permutations testées en ensemble, la comparaison prend en compte toutes les permutations possibles. Dans notre cas, soit deux combinaisons A et B et N_R la nombre de nœuds distincts dans le cluster ainsi que N_{PT} le nombre de processus terminaux de l'expérimentation, alors :

$$A \equiv B \pmod{N_R} \text{ si } |A| = |B| = N_{PT} \mid \forall b \in B, \exists a \in A \text{ tq. } a \equiv b \pmod{N_R}$$

Lors de la transformation en ensemble des combinaisons testées, il faut s'assurer que la taille de l'ensemble résultant reste de valeur N_{PT} . Nous ne pouvons pas simplement réduire chacun de ces éléments au maximum, sinon la combinaison $\{0, 1, 3\}$ dans notre exemple deviendrait l'ensemble $\{0, 1\}$ et l'ensemble perd un élément. L'algorithme 3 permet cette transformation. Chaque élément est à tour de rôle réduit au maximum et inséré dans un ensemble. Si lors de l'insertion, nous découvrons qu'un élément est déjà présent dans l'ensemble, nous l'additionnons avec N_R jusqu'à pouvoir l'insérer. Ceci garantit que chaque combinaison, lors de sa transformation en ensemble, garde la bonne taille en ayant la plus petite réduction possible. Cette technique permet d'éviter de tester deux combinaisons équivalentes.

Algorithm 3 Transformation d'une combinaison en ensemble

```

procedure comb_to_set(comb, NR)
    A ← HashSet :: new()
    NPT ← comb.len
    for i ← 0 to NPT do
        a ← comb[i] mod NR
        while !A.insert(a) do
            a ← a + NR
        end while
        i ← i + 1
    end for
    return A
end procedure

```

Cette transformation des combinaisons en ensembles permet ensuite à l'orchestrateur de garder un ensemble de combinaisons testées. Il s'agit d'un ensemble d'ensembles. Lors de la génération d'une nouvelle combinaison, elle est alors transformée en ensemble et l'orchestrateur vérifie si elle est présente dans l'ensemble des combinaisons. Cette vérification est efficace si les ensembles sont représentés par des HashSet.

5.5 Épinglage de l'EmulCore par l'EManager

Comme exprimé dans les précédents chapitres, l'application Kollaps fonctionne dans un environnement asynchrone. Lorsqu'un EManager reçoit l'ordre de mettre en place une expérimentation, il démarre un EmulCore. Afin que l'EmulCore fonctionne en concurrence avec l'EManager et les autres composants, l'EManager crée une coroutine pour chaque EmulCore.

La gestion des coroutines est déléguée à la librairie Tokio qui implémente un scheduler. Ce scheduler a à sa disposition une réserve de threads (*thread pool*). Il se charge de répartir les coroutines sur les différents threads. Il se peut qu'une coroutine soit balancée d'un thread à un autre. Chaque méthode déclarée comme asynchrone est considérée comme une coroutine. Dès lors, durant l'exécution d'une suite d'instructions, si un appel à une méthode asynchrone est effectué, l'exécution est mise en pause jusqu'à la terminaison de l'appel. C'est à ce moment-là que la coroutine qui contient l'exécution appelante peut être retirée de son thread pour être réinsérée sur un autre une fois l'exécution de la méthode appelée terminée.

Il faut préciser que pour appeler une méthode asynchrone, l'exécution appelante doit aussi être dans un environnement asynchrone, donc dans une coroutine. Si nous remontons jusqu'au premier appel, cela implique que l'exécution du *main* se fait à l'intérieur d'une coroutine. Comme l'ensemble du code se déroule dans un environnement asynchrone, chaque instruction est exécutée à l'intérieur d'une coroutine.

Toutes les structures appartiennent à la coroutine dans laquelle elles ont été créées et sont déplacées avec la coroutine de thread en thread. Ces structures font partie du contexte de la coroutine.

Dans l'EmulCore, il existe plusieurs fonctions internes qui sont définies comme asynchrones. Certaines de ces méthodes nécessitent d'avoir accès à des références d'éléments présents dans l'EmulCore. Le problème survient lorsqu'un EmulCore fait appel à une de ces méthodes. Si l'EmulCore fournit à une fonction une référence d'un élément qu'il possède, cette référence ne peut pas être garantie comme valide. Effectivement, comme l'EmulCore est mis en pause et potentiellement déplacé vers un autre thread lors de l'appel à la fonction, la fonction se retrouvera avec une référence pointant vers l'ancien endroit de l'EmulCore, sur le mauvais thread.

Ce problème peut être résolu de deux façons différentes. La première consiste à ne pas passer de références, mais de créer des clones des données avant de les transmettre par valeur. Il y a aussi la possibilité de donner la propriété des éléments à la fonction. Cette première solution se base sur le passage par valeur.

La deuxième, qui est utilisée et recommandée, consiste à épingle une structure à un endroit dans la mémoire. C'est ce qui est appelé *Pin* en Rust. Typiquement, lors de la création d'un EmulCore, l'EManager va épingle la structure à un endroit dans le tas (*heap*). De cette manière, l'EmulCore n'appartient à aucune coroutine directement. La coroutine dans laquelle l'EmulCore a été créé et épingle possède uniquement une référence vers l'endroit du tas où est enregistré l'EmulCore. Cela demande qu'à chaque appel de méthode sur l'EmulCore, le code doit d'abord récupérer l'EmulCore sur le tas en procédant à ce qui est appelé un *UnPin*. Cette façon de procéder permet ainsi de ne pas devoir dupliquer des données ou de devoir donner la propriété à d'autres fonctions.

5.6 EmulCore nettoie en toutes circonstances

Dans le cas où un EmulCore doit gérer le cycle de vie d'un ou plusieurs PTs, il se sert de la librairie DockHelper pour les démarrer au sein d'un conteneur. Aussi, l'EmulCore est responsable du cycle de vie des Reporters.

Cela signifie que si un problème intervient soit dans l'EmulCore, soit dans l'EManager, il est nécessaire d'avoir une procédure pour arrêter les conteneurs et stopper les Reporter. Sinon, nous pouvons nous retrouver avec des processus orphelins sur le système.

Rust nous permet d'implémenter le trait *Drop*. Ce trait contient qu'une seule méthode qui est appelée lorsqu'une structure n'a plus de propriétaire et est donc éliminée.

L'idée est proche de l'implémentation d'un destructeur en C++ mais diffère dû à la nature du fonctionnement de Rust.

Le Reporter implémente le trait *Drop* et dans son implémentation, il supprime la liaison Unix qui lui permet de communiquer avec l'EmulCore. En supprimant intentionnellement la liaison Unix, celle-ci va à son tour supprimer le socket sur lequel elle se repose.

L'EmulCore implémente aussi le trait *Drop*. Dans son implémentation, il lance une nouvelle coroutine qui va s'occuper d'arrêter les conteneurs qui sont gérés par l'EmulCore.

L'arrêt des Reporters se fait automatiquement. Les Reporters sont lancés avec l'utilisation de la méthode *Spawn* sur une commande. Lors de la création de la commande, il est possible d'indiquer l'attribut *kill_on_drop* qui tue le processus enfant dès qu'il n'appartient à plus personne.

Le trait *Drop* fait partie de la librairie standard, ce qui signifie que son exécution est synchrone. Les fonctions pour arrêter les conteneurs sont asynchrones. C'est pour cela que l'EmulCore doit lancer une nouvelle coroutine depuis son implémentation du *Drop*. Cela permet aussi de faire un nettoyage en arrière-plan sans impacter l'exécution du programme.

En utilisant ce trait, nous nous assurons que lors d'une erreur, nous ne produisons pas de processus orphelins.

5.7 Accès parallèles aux statistiques dans le Reporter

Comme indiqué dans le chapitre 4.9.1, un Reporter utilise un HashMap afin d'enregistrer et de surveiller les changements d'utilisation de bande passante.

Cet HashMap subit des accès parallèles en écriture et en lecture. Au lieu d'utiliser l'implémentation standard du HashMap et de le protéger par un Mutex, le programme utilise une implémentation *lock-free*. Cette implémentation provient de la librairie [lockfree](#)²³.

Dans notre cas, nous avons deux threads qui modifient l'HashMap en parallèle. [Leurs tests](#)²⁴ indiquent que pour deux threads, leur implémentation offre un avantage 24% d'accès concurrents supplémentaires par seconde par rapport à la version avec Mutex.

5.8 Améliorations du Monitor

Le programme eBPF était déjà utilisé dans l'ancienne version de Kollaps. Cette nouvelle version reprend ce programme en l'améliorant et en corrigeant certains problèmes.

Le premier problème était l'incompatibilité du programme avec les versions du Kernel Linux 6.0 ou supérieures. Effectivement, dans la version 6.0, les contributeurs du Kernel

²³<https://crates.io/crates/lockfree>

²⁴<https://docs.rs/crate/lockfree/latest/source/BENCHMARKS.md>

ont restructuré la structure représentant l'[entête IP](#)²⁵. Les champs *saddr* et *daddr* représentant l'adresse IP source, respectivement l'adresse IP de destination, ne sont plus directement accessibles, alors que c'était le cas dans les Kernels de version [5.x](#)²⁶. Ils ont été remplacés par une [union anonyme](#)²⁷. La première correction fût de modifier la façon de récupérer ces champs. Il est aussi impossible de se baser sur le début de cette union anonyme, sinon le programme devient incompatible avec les Kernels de versions inférieures à 6.0. Afin d'être compatible avec toutes les versions, le point d'ancrage choisi est le champ TTL. Nous pouvons donc récupérer l'adresse source et l'adresse de destination en retrouvant le décalage du champ TTL et en y ajoutant un décalage de 64 bits[15].

La deuxième correction importante est la suppression de la vérification du protocole. Dans l'ancienne version, le programme vérifiait quel protocole de transport le paquet utilisait. Si le protocole ne correspondait pas à TCP, le paquet était ignoré. Cette erreur empêchait Kollaps de fonctionner correctement et empêchait Kollaps d'être protocole-agnostique.

L'amélioration principale du programme eBPF se trouve dans sa façon de fonctionner. Auparavant, le programme eBPF ajoutait simplement dans un tableau la quantité d'octets envoyés à une adresse, ce qui produisait une valeur cumulative. Le programme dans l'espace utilisateur devait donc traiter cette valeur cumulative et calculer le débit instantané par destination. Pour ce faire, l'ancienne version du programme dans l'espace utilisateur prenait le temps actuel via un appel système et commençait son parcours des valeurs. Durant le parcours, il calculait la différence entre l'ancienne valeur pour une adresse de destination et la nouvelle valeur. Cette différence indique alors le nombre d'octets qui ont été transférés vers cette destination entre deux parcours de la structure. En fonction du débit trouvé, il exécutait différentes tâches. L'ancien programme en espace utilisateur fonctionnait aussi dans un environnement asynchrone.

Comme nous l'avons vu dans le chapitre 5.5, lorsqu'une coroutines appelle d'autres fonctions asynchrones, la coroutines est mise en pause jusqu'à ce que les fonctions appelées soient terminées. Donc, durant son parcours, pour chaque entrée dans la structure, le programme doit calculer la différence d'octets, calculer le débit et potentiellement exécuter des tâches supplémentaires qui mettront sa coroutines en pause. Le temps de traitement de chaque entrée dans la structure varie alors beaucoup.

Plus il y a d'entrées, plus le temps écoulé entre la prise du temps initial via l'appel système et le traitement de la dernière entrée de la structure est grand. Cela provoque un calcul du débit instantané faussé pour les dernières entrées de la structure. Le problème pourrait être amoindri si la prise de temps se produisait à chaque passage d'une entrée à une autre lors du parcours, mais demanderait beaucoup d'appels système. De plus, l'utilisation d'une valeur cumulative provoque des dépassements d'entiers qui doivent être gérés dans l'espace utilisateur si beaucoup d'octets sont envoyés à une même destination.

La nouvelle version a pour but de corriger ces problèmes en ne calculant plus le débit instantané dans l'espace utilisateur, mais en le faisant directement dans le Monitor, dans l'espace noyau. Cette nouvelle version du Monitor transmet directement le débit instantané pour une destination. Pour ce faire, il enregistre par destination la totalité des octets qui sont envoyés, ainsi que le temps auquel la dernière mise à jour concernant cette destination a été envoyée au programme utilisateur.

Lors du traitement d'un paquet, il vérifie que la dernière mise à jour envoyée au programme en espace utilisateur concernant la destination du paquet s'est produite il y a plus de 25 millisecondes. Si c'est le cas, alors il calcule le débit instantané pour la destination et remet son compteur à zéro. Dans le cas où la mise à jour est assez récente, il incrémentera juste son compteur.

²⁵<https://www.rfc-editor.org/rfc/rfc791#section-3.1>

²⁶<https://elixir.bootlin.com/linux/v5.19.17/source/include/uapi/linux/ip.h#L86>

²⁷<https://elixir.bootlin.com/linux/v6.0.17/source/include/uapi/linux/ip.h#L86>

Il est considéré qu'il est impossible de produire un dépassement d'entier non signé de 64 bits en 25 millisecondes. Cela signifierait qu'il faudrait produire $\frac{2^{64} \cdot 1000}{25}$ octets par seconde, soit 737.9 Exabytes/s si le compteur de 64 bits décrit le nombre d'octets. Si aucun paquet en direction d'une destination n'est produit, alors aucun événement n'est produit par le Monitor. C'est au programme dans l'espace utilisateur de considérer un débit comme revenu à zéro si aucune mise à jour n'a été reçue depuis un certain temps.

Cette nouvelle version enlève les problèmes de calcul de débit dans l'espace utilisateur et rend le code plus *élégant*. Il permet d'apporter une meilleure précision et bénéficie de l'exécution privilégiée de l'eBPF fonctionnant dans l'espace Kernel.

De plus, à travers les différentes améliorations, le nouveau programme ajoute uniquement une division supplémentaire toutes les 25 millisecondes et garde donc sa complexité initiale.

5.9 Général

Ce chapitre compile plusieurs éléments d'implémentation qui ne sont pas liés directement à un composant ou une structure, mais qui concernent le programme en général.

5.9.1 Utilisation de Tokio et asynchrone

Comme déjà abordé dans d'autres chapitres, le programme fonctionne sur un système de coroutines. Aucun système de gestion de coroutine n'est présent directement dans la librairie standard de Rust. Rust repose sur des implémentations externes de gestionnaires, tout en intégrant dans le langage des mots clés et des traits qui permettent d'unifier les implémentations asynchrones. Typiquement, les mots clés `async` et `await` sont compris dans Rust.

Le projet utilise la librairie Tokio comme gestionnaire de coroutines. Il s'agit d'une des librairies les plus utilisées dans ce domaine.

5.9.2 Instanciation des composants, le Runner

Lors de la conception (ch. 4), nous avons vu que le programme est divisé en plusieurs composants. Ces composants fonctionnent en concurrence, c'est pourquoi il faut une entité supérieure qui se charge de les lancer. Cette entité est le Runner.

Le Runner est le seul paquet du programme Kollaps qui produit un exécutable. Tous les autres paquets produisent des librairies. La tâche du Runner est de récupérer les configurations utilisateur et de vérifier la présence des logiciels nécessaires (`nsenter`, `docker`, etc.) pour ensuite démarrer le CManager et l'EManager. Dans le cas où le nœud est un leader, c'est l'EManager qui s'occupe de démarrer l'OManager.

Le seul autre paquet qui produit un binaire ne fait complètement partie du programme Kollaps. Il s'agit du Reporter. Le paquet Reporter produit son propre binaire qui peut ensuite être utilisé par les EmulCores.

5.9.3 Rust et safe code

Rust garantit que notre code est *safe* grâce au Borrow Checker et en limitant certaines fonctionnalités que nous pourrions retrouver dans des langages comme C ou C++. La sûreté d'un code est vérifiée au moment de la compilation. De ce fait, l'analyse statique de Rust est de nature conservatrice. C'est pourquoi il existe la notation `unsafe`.

Le passage en mode `unsafe` nous permet notamment de :

- Déréférencer des *raw pointers*.

- Appeler d'autres fonctions non sûres.
- Accéder ou modifier des variables statiques mutables.
- Implémenter des traits non sûrs.
- Accéder à des attributs d'unions.

Dans la plupart des cas, il faut éviter d'utiliser des parties de code non sûres. La majorité des utilisations de code non sûr est fait lors de communication avec des bibliothèques écrites en C ou autres langages.

Dans la précédente version de Kollaps, certaines parties de code étaient implémentées de manière non sûre, par commodité. Comme ce projet propose une réécriture complète du code source, ces parties ont été supprimées.

Un exemple typique est la transformation des adresses IP en entiers de 32 bits, requis par la bibliothèque TCAL. L'ancienne implémentation considérait que les adresses IP étaient stockées sous forme d'un tableau avec le format suivant : pour l'adresse 192.168.1.1, nous avions dans la mémoire [1, 1, 168, 192] avec la première case à l'adresse basse et la dernière à l'adresse haute (Little Endian). Ensuite, il utilisait la méthode non sûre `std::mem::transmute`²⁸ de la bibliothèque standard pour récupérer ces quatre cases de 8 bits en un entier de 32.

La documentation de *transmute* dit :

This makes *transmute* **incredibly unsafe**. *transmute* should be the absolute last resort.

Lors de la réécriture, il a été choisi d'utiliser l'implémentation de la bibliothèque standard `IpAddr` pour représenter une adresse IP. Ceci permet une meilleure opérabilité à travers le système. Lors des premiers changements, la bibliothèque TCAL semble ne plus fonctionner. Lors de la conversion d'une `IpAddr` en tableau de bytes pour ensuite être converti en entier de 32 bits, le résultat n'est plus le même qu'auparavant. Le bug apparaît à cause de la façon dont les données sont stockées. Le TCAL semble s'attendre à des entiers représentant des adresses IP en mode Little Endian (ce qui est décrit nulle part). Lorsque nous demandons à une `IpAddr` de nous rendre le tableau de bytes représentant l'adresse, celui-ci nous le donne en Big Endian ([192, 168, 1, 1]). L'utilisation de la méthode *transmute* fonctionne uniquement si la machine travaille en mode Little Endian, ce qui est le cas pour la plupart des processeurs aujourd'hui[18]. Il s'agit sûrement de la raison pour laquelle le problème n'a pas été détecté avant.

C'est à ces moments-là où nous pouvons constater les dégâts que peut causer l'utilisation de code non sûr. Le problème ne vient pas du changement du code pour l'utilisation de l'`IpAddr`, mais bel est bien de l'utilisation de code non sûr. La nouvelle implémentation fonctionne en mode *safe* et donne le même résultat peu importe le boutisme utilisé.

5.9.4 Division par paquets indépendants

Rust encourage une division du code en paquets. C'est ce qui est fait dans le programme. Chaque paquet produit sa propre bibliothèque qui peut être ensuite intégrée par d'autres paquets. Ce principe de fonctionnement par bibliothèques permet de facilement définir les dépendances de chaque paquet et de compiler et construire chaque paquet indépendamment des autres. L'ensemble des paquets forme un espace de travail (*Workspace*). Nous pouvons soit décider de compiler tous les paquets d'un espace de travail, ou uniquement certains.

Chacun des paquets produit soit une bibliothèque, soit un exécutable. Dans notre cas, deux exécutables sont créés : le Reporter et le Runner.

²⁸<https://doc.rust-lang.org/stable/std/mem/fn.transmute.html>

5.9.5 SymMatrix

Dans le projet, deux structures représentent des graphes. La première est le CGraph (voir ch. 4.5) et la deuxième est le Network (voir annexes ch. 11.4). Les deux structures représentent des graphes non dirigés. Les matrices d'adjacence des graphes non dirigés sont des matrices symétriques.

Chaque élément d'une matrice symétrique est dupliqué (sauf pour la diagonale). Comme ces deux structures doivent être envoyées sur le réseau, il est intéressant de ne pas stocker deux fois la même information et gagner de la place en implémentant une structure représentant une matrice symétrique. C'est exactement ce qui a été fait pour ce projet. La structure s'appelle *SymMatrix*.

L'implémentation d'une telle structure apporte quelques difficultés.

Tout d'abord, dans notre implémentation, nous nous imaginons utiliser uniquement le triangle supérieur de la matrice, comme le montre la Figure 27. Les données sont sauvegardées dans un vecteur. L'exemple présenté sur la Figure sera utilisé tout au long de l'explication.

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
	4 (1,1)	5 (1,2)	6 (1,3)
		7 (2,2)	8 (2,3)
			9 (3,3)

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)	4 (1,1)	5 (1,2)	6 (1,3)	7 (2,2)	8 (2,3)	9 (3,3)
------------	------------	------------	------------	------------	------------	------------	------------	------------	------------

Figure 27: Représentation du triangle supérieur d'une matrice

La première chose à considérer est la création d'une telle matrice. La taille du vecteur de valeurs nécessaire est égal au résultat d'une suite arithmétique de raison 1 : nous avons, en partant depuis la première ligne :

$$n + (n - 1) + (n - 2) + \cdots + 1$$

éléments. La formule est donc pour une matrice de taille n :

$$n \cdot \frac{n + 1}{2}$$

Ensuite, vient la partie d'adressage. Nous voulons accéder à un élément de la matrice en donnant la ligne et la colonne. Dans le cas d'une matrice symétrique, l'indexation

$(i, j) = (j, i)$. En considérant uniquement le triangle supérieur, nous prenons toujours le plus petit indice comme la ligne et le plus grand comme la colonne. Cette technique fera toujours pointer les indices dans le triangle supérieur. Il faut ensuite un moyen de convertir un indice de type (i, j) en indice dans le tableau de valeur. La technique est de calculer combien d'éléments il faut parcourir dans la matrice avant d'accéder à la ligne i . Ceci donne le décalage dans le tableau des valeurs pour accéder à la bonne ligne. Dans l'exemple présenté, si $i = 2$ (troisième ligne), alors les éléments de la troisième ligne commenceront à l'indice 7 et termineront à l'indice 8.

Pour connaître cette information, nous pouvons suivre le même raisonnement que précédemment. La somme des éléments constituant les lignes précédentes est la somme de la suite arithmétique partant de n , allant jusqu'à $n + (i - 1) \cdot r$ et de raison $r = -1$. La formule suivante indique alors le nombre d'éléments constituant les lignes précédentes :

$$x = i \cdot \frac{n + (n + (i - 1) \cdot r)}{2}$$

La valeur x indique l'indice de départ dans le tableau pour la ligne i . Il suffit maintenant de trouver le décalage au niveau de la colonne. Ce décalage peut facilement être trouvé en soustrayant la colonne par la ligne. Nous avons donc

$$y = j - i$$

Finalement, après avoir calculé x et y , l'indice du tableau de valeur comportant la case (i, j) est donné par $x + y$.

L'implémentation de la matrice offre aussi la possibilité d'appliquer une fonction à chacun des éléments de la matrice. Pour ce faire, l'implémentation offre une méthode prenant en paramètre une *Closure* qui est ensuite exécutée pour chaque élément du tableau de valeurs. La *Closure* reçoit en paramètre la ligne et la colonne qui est en cours de traitement afin de pouvoir exécuter des instructions en fonction. Afin de fournir à la *Closure* la ligne et la colonne en cours de traitement, l'implémentation doit être capable de faire le travail d'indexation inverse. C'est-à-dire transformer un indice du tableau en une paire (ligne, colonne). La transformation commence à rechercher dans quelle ligne l'indice actuel se trouve. Ici, il n'existe pas de formule. Il est nécessaire d'addition la taille de chaque ligne une à une jusqu'à trouver que l'index courant se trouve entre la somme actuelle et la somme actuelle + la taille de la ligne actuelle. En prenant notre exemple, si l'indice courant est $i = 7$, l'algorithme calcule ceci :

$$0 \leq 7 < 4? \Rightarrow \text{false}$$

$$4 \leq 7 < 7? \Rightarrow \text{false}$$

$$7 \leq 7 < 9? \Rightarrow \text{true}$$

l'indice de la ligne est donc 2. Nous nommons cet indice r .

Après avoir trouvé la ligne, il est facile de retrouver l'indice de la colonne en réutilisant la fonction utilisée pour trouver le décalage de la colonne lors de la transformation inverse :

$$y = c - i \Leftrightarrow c = y + i \Leftrightarrow c = (i - r) + i$$

Le résultat final est la paire (r, c) qui représente (ligne, colonne).

Ce chapitre termine la partie dédiée à l'implémentation. La plupart de ces implémentations indépendantes ont leur propre série de tests unitaires. Tester des parties d'implémentation indépendamment les unes des autres ne suffit pas à garantir le fonctionnement global du système. Le chapitre suivant discute et montre les résultats de tests d'intégration effectués sur cette nouvelle version de Kollaps.

6 Tests

Les tests se focalisent sur les nouvelles fonctionnalités qu'apporte le projet. Il s'agit de la création d'un cluster Kollaps, de l'orchestration des expérimentations et la robustesse générale de cette nouvelle version.

Les tests présentés sont des tests d'intégration. Les différentes parties critiques, comme le CGraph et l'orchestrateur, ont leur série de tests unitaires qui permettent de vérifier que leur implémentation est conforme à ce qui est présenté dans ce document.

Tout d'abord, le chapitre expose l'environnement de test puis définit huit scénarios, répartis en plusieurs catégories, qui permettent de tester dans l'ensemble les nouvelles fonctionnalités.

6.1 Environnement de test

L'ensemble des tests se déroulent sur le même environnement. Cet environnement est constitué de trois machines virtuelles, connectées via un réseau simulé de type NAT.

L'hyperviseur utilisé est [Gnome Boxes](#)²⁹ version 43.2-stable qui est une interface pour l'émulateur [Qemu](#)³⁰ version 7.0.0.

Les trois machines virtuelles sont des instances d'Ubuntu 20.04 avec ces propriétés :

- Kernel : 5.15.0-58-generic
- Rust : 1.59
- Docker : 23.0.0
- LLVM : 13

Ces machines forment le cluster Kollaps. Le schéma réseau ci-dessous (Figure 28) montre leurs propriétés IP.

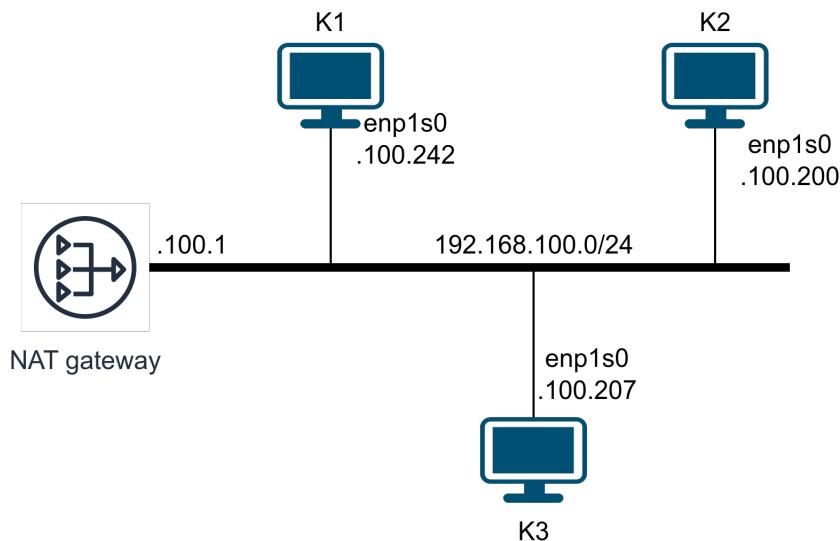


Figure 28: Environnement réseau des machines du cluster

²⁹<https://help.gnome.org/users/gnome-boxes/stable/>

³⁰<https://www.qemu.org/>

L'expérimentation utilisée pour confirmer le fonctionnement de cette implémentation est représentée sur la Figure 29. C1, C2 et C3 rejoignent simultanément l'expérimentation au début de celle-ci et la quittent aussi en même temps au bout de 300 secondes.

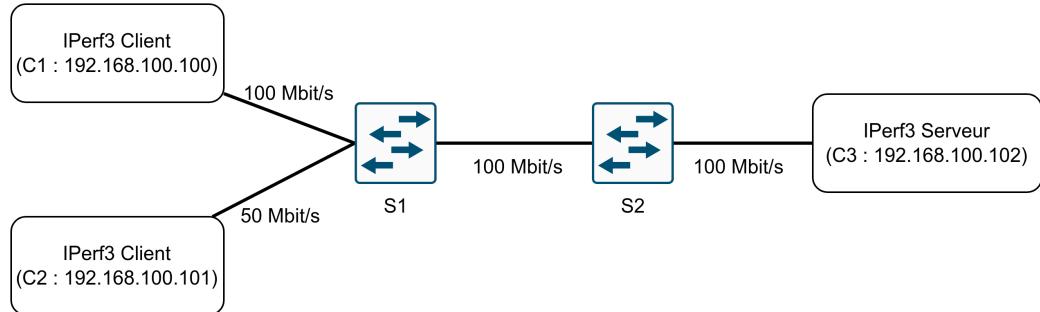


Figure 29: Expérimentation utilisée pour les tests

Cette expérimentation est utilisée uniquement dans la seconde catégorie des tests. La première catégorie est la gestion du cluster.

6.2 Gestion du cluster

Les scénarios suivants ont pour but de vérifier les différentes situations qui impactent la création et la gestion d'un cluster Kollaps. Deux sous-catégories sont définies :

- Adhésion : lorsqu'un nœud s'ajoute au cluster.
- Retrait et redémarrage : lorsqu'un nœud s'arrête ou subit un redémarrage.

Chaque nom de machine fait référence au schéma réseau montré sur la Figure 28. De manière générale, sauf explicitement écrit, le leader du cluster est le nœud K1.

La première sous-catégorie abordée est l'adhésion.

6.2.1 Adhésion

Scénario 1

Objectif : Lors de la présence d'un cluster stable, sans activité, un nœud doit être accepté lorsqu'une demande d'adhésion est reçue par le leader. Ce nœud exécute les tests de performance nécessaires et à la fin de la procédure d'adhésion, celui-ci doit être pris en compte pour les futurs déploiements.

Scénario : Lancé Kollaps en mode leader sur K1. Lancer une nouvelle instance de Kollaps en mode non-leader sur la machine K2. Attendre la fin de la procédure d'adhésion pour K2 et lancer une autre instance de Kollaps en mode non-leader sur K3.

L'exécution de ce scénario apporte les résultats attendus. La Figure 30 ci-dessous (journalisation de K1), montre les deux nœuds K2 et K3 qui rejoignent le cluster séquentiellement. La partie bleue est l'adhésion de K2, qui a exécuté un test avec le leader, et la partie rouge est l'adhésion de K3 qui a exécuté un test sur K2. Nous avons l'orchestrateur qui, à la fin de chaque procédure d'adhésion, prend en compte le nouveau nœud.

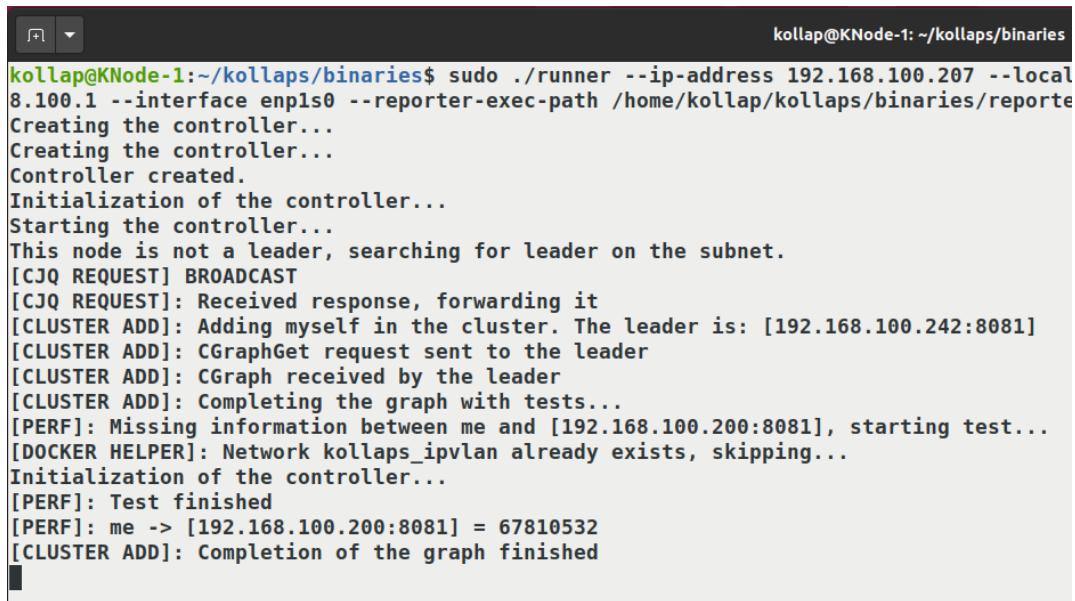
```

kollap@KNode-1:~/kollaps/binaries$ sudo ./runner --ip-address 192.168.100.242 --local-spy 192.168.100.1 --interface empl1s0 --leader --reporter-exec-path /home/kollap/kollaps/binaries
[sudo] password for kollap:
Creating the controller...
Creating the controller...
Controller created.
Initialization of the controller...
Starting the controller...
[CREATE CLUSTER]: Creating a cluster
[DOCKER HELPER]: Network kollaps_ipvlan already exists, skipping...
Initialization of the controller...
[CJQ REQUEST]: Accept request
[PERF]: Received perf request, starting the server...
[PERF]: Server started
-----
Server listening on 5201
-----
Accepted connection from 192.168.100.200, port 40374
[ 5] local 192.168.100.242 port 5201 connected to 192.168.100.200 port 40390
[ ID] Interval           Transfer      Bitrate
[ 5]  0.00-1.00   sec  7.46 GBytes  64.1 Gbits/sec
[ 5]  1.00-2.00   sec  8.50 GBytes  73.0 Gbits/sec
[ 5]  2.00-3.00   sec  8.68 GBytes  74.6 Gbits/sec
[ 5]  3.00-4.00   sec  9.01 GBytes  77.4 Gbits/sec
[ 5]  4.00-5.00   sec  8.74 GBytes  75.1 Gbits/sec
[ 5]  5.00-6.00   sec  8.72 GBytes  74.9 Gbits/sec
[ 5]  6.00-7.00   sec  8.81 GBytes  75.7 Gbits/sec
[ 5]  7.00-8.00   sec  8.94 GBytes  76.8 Gbits/sec
[ 5]  8.00-9.00   sec  9.00 GBytes  77.3 Gbits/sec
[ 5]  9.00-10.00  sec  8.83 GBytes  75.9 Gbits/sec
[ 5] 10.00-11.00  sec  8.73 GBytes  75.0 Gbits/sec
[ 5] 11.00-12.00  sec  8.37 GBytes  71.9 Gbits/sec
[ 5] 12.00-13.00  sec  8.85 GBytes  76.0 Gbits/sec
[ 5] 13.00-14.00  sec  8.77 GBytes  75.4 Gbits/sec
[ 5] 14.00-15.00  sec  8.42 GBytes  72.3 Gbits/sec
[ 5] 15.00-15.04  sec  357 MBytes  70.0 Gbits/sec
-----
[ ID] Interval           Transfer      Bitrate
[ 5]  0.00-15.04  sec  130 GBytes  74.3 Gbits/sec          receiver
[TCPLISTENER]: Connection closed by the peer: 192.168.100.200:42097
[TCPLISTENER]: Connection closed by the peer: 192.168.100.200:39337
[HEARTBEAT CHECK]: Cannot send receive heartbeat, it must have been aborted
[TCPLISTENER]: Connection closed by the peer: 192.168.100.200:44833
[Orchestrator] : Updating with CGraph
[Orchestrator] : Old Residual Graph Size : 1
[Orchestrator] : To Remove : []
[Orchestrator] : To Add : [ClusterNodeInfo { ip_addr: 192.168.100.200, port: 8082 }]
[Orchestrator] : UUID Affected by removes : []
[Orchestrator] : New Residual Graph Size : 2
-----
[CJQ REQUEST]: Accept request
[TCPLISTENER]: Connection closed by the peer: 192.168.100.207:39133
[TCPLISTENER]: Connection closed by the peer: 192.168.100.207:37451
[HEARTBEAT CHECK]: Cannot send receive heartbeat, it must have been aborted
[Orchestrator] : Updating with CGraph
[Orchestrator] : Old Residual Graph Size : 2
[Orchestrator] : To Remove : []
[Orchestrator] : To Add : [ClusterNodeInfo { ip_addr: 192.168.100.207, port: 8082 }]
[Orchestrator] : UUID Affected by removes : []
[Orchestrator] : New Residual Graph Size : 3

```

Figure 30: Journalisation de K1 (leader) représentant l'adhésion de deux nouveaux nœuds consécutivement.

La capture de sortie de K3 (Figure 31) montre que lorsque la requête d'un nœud est acceptée, le nœud récupère le CGraph sur le leader et exécute les tests de performance manquants.



```

kollap@KNode-1:~/kollaps/binaries$ sudo ./runner --ip-address 192.168.100.207 --local
8.100.1 --interface enp1s0 --reporter-exec-path /home/kollap/kollaps/binaries/reporter
Creating the controller...
Creating the controller...
Controller created.
Initialization of the controller...
Starting the controller...
This node is not a leader, searching for leader on the subnet.
[CJQ REQUEST] BROADCAST
[CJQ REQUEST]: Received response, forwarding it
[CLUSTER ADD]: Adding myself in the cluster. The leader is: [192.168.100.242:8081]
[CLUSTER ADD]: CGraphGet request sent to the leader
[CLUSTER ADD]: CGraph received by the leader
[CLUSTER ADD]: Completing the graph with tests...
[PERF]: Missing information between me and [192.168.100.200:8081], starting test...
[DOCKER HELPER]: Network kollaps_ipvlan already exists, skipping...
Initialization of the controller...
[PERF]: Test finished
[PERF]: me -> [192.168.100.200:8081] = 67810532
[CLUSTER ADD]: Completion of the graph finished

```

Figure 31: Journalisation de K3 lors de l'adhésion au cluster formé par K1 et K2.

Scénario 2

Objectif : Lorsqu'une procédure d'adhésion est en cours et qu'un autre nœud envoie une requête d'adhésion, celle-ci doit être refusée et le nœud doit attendre un temps défini par le leader.

Scénario : Créer un cluster formé uniquement du leader K1. Lancer l'application Kollaps en non-leader sur K2, attendre que la requête soit acceptée par K1. Durant le test de performance que doit exécuter K2, lancer l'application en non-leader sur K3.

L'exécution de ce scénario donne les résultats attendus. Sur la journalisation de K1 (Figure 32), nous pouvons observer en bleu la procédure d'adhésion de K2 qui réussit. Au milieu de cette procédure, K1 reçoit la requête de K3 et la refuse (rectangle rouge à l'intérieur du rectangle bleu). Puis, à la suite de la procédure d'adhésion de K2, K3 réessaie et réussit à rejoindre le cluster (rectangle rouge).

```

Accepted connection from 192.168.100.200, port 39110
[ 5] local 192.168.100.242 port 5201 connected to 192.168.100.200 port 39120
[ ID] Interval      Transfer     Bitrate
[ 5]  0.00-1.00   sec  6.13 GBytes  52.7 Gbits/sec
[ 5]  1.00-2.00   sec  5.42 GBytes  46.5 Gbits/sec
[ 5]  2.00-3.00   sec  8.55 GBytes  73.4 Gbits/sec
[ 5]  3.00-4.00   sec  9.03 GBytes  77.6 Gbits/sec
[ 5]  4.00-5.00   sec  7.52 GBytes  64.6 Gbits/sec
[ 5]  5.00-6.00   sec  118 GBytes  67.6 Gbits/sec
[ 5]  6.00-7.00   sec  7.75 GBytes  66.6 Gbits/sec
[ 5]  7.00-8.00   sec  7.87 GBytes  67.6 Gbits/sec
[ 5]  8.00-9.00   sec  7.75 GBytes  66.6 Gbits/sec
[ 5]  9.00-10.00  sec  7.80 GBytes  67.0 Gbits/sec
[ 5]  10.00-11.00  sec  8.96 GBytes  77.0 Gbits/sec
[ 5]  11.00-12.00  sec  8.94 GBytes  76.8 Gbits/sec
[ 5]  12.00-13.00  sec  8.93 GBytes  76.7 Gbits/sec
[ 5]  13.00-14.00  sec  8.07 GBytes  69.3 Gbits/sec
[ 5]  14.00-15.00  sec  7.63 GBytes  65.6 Gbits/sec
[ 5]  15.00-15.05  sec  180 MBytes  31.6 Gbits/sec
[ ID] Interval      Transfer     Bitrate
[ 5]  0.00-15.05  sec  118 GBytes  67.6 Gbits/sec
[TCPLListener]: Connection closed by the peer: 192.168.100.200:37407           receiver
[TCPLListener]: Connection closed by the peer: 192.168.100.200:43947
[TCPLListener]: Connection closed by the peer: 192.168.100.200:40725
[Orchestrator] : Updating with CGraph
[Orchestrator] : Old Residual Graph Size : 1
[Orchestrator] : To Remove : []
[Orchestrator] : To Add : [ClusterNodeInfo { ip_addr: 192.168.100.200, port: 8082 }]
[Orchestrator] : UUID Affected by removes : []
[Orchestrator] : New Residual Graph Size : 2
[HEARTBEAT CHECK]: Cannot send receive heartbeat, it must have been aborted
[TCPLListener]: Connection closed by the peer: 192.168.100.207:36723
[Orchestrator] : Updating with CGraph
[Orchestrator] : Old Residual Graph Size : 2
[Orchestrator] : To Remove : []
[Orchestrator] : To Add : [ClusterNodeInfo { ip_addr: 192.168.100.207, port: 8082 }]
[Orchestrator] : UUID Affected by removes : []
[Orchestrator] : New Residual Graph Size : 3

```

Figure 32: Journalisation de K1 lors de l'ajout simultané de K2 et K3.

La journalisation de K3 ci-dessous (Figure 33) montre qu'il a bien reçus la demande d'attente du leader.

```

kollap@KNode-1:~/kollaps/binaries$ sudo ./runner --ip-address 192.168.100.207 --local
8.100.1 --interface enp1s0 --reporter-exec-path /home/kollap/kollaps/binaries/reporter
Creating the controller...
Creating the controller...
Controller created.
Initialization of the controller...
Starting the controller...
[CLUSTER ADD] Adding myself in the cluster. The leader is: [192.168.100.242:8081]
[CLUSTER ADD] CGraphGet request sent to the leader
[CLUSTER ADD] CGraph received by the leader
[CLUSTER ADD] Completing the graph with tests...
[PERF]: Missing information between me and [192.168.100.200:8081], starting test...
[PERF]: Test finished
[PERF]: me -> [192.168.100.200:8081] = 61325747
[CLUSTER ADD]: Completion of the graph finished

```

Figure 33: Journalisation de K3 lors de la tentative d'adhésion en simultané avec K2.

Scénario 3

Objectif : Lors d'une procédure d'adhésion, si le nœud essayant de rejoindre le cluster subit des erreurs, il doit envoyer un message d'annulation de procédure au leader. Le nœud ayant subi une erreur réessaie ensuite de rentrer dans le cluster.

Scénario : Créer un cluster stable formé de K1 et K3. Arrêter le processus Kollaps sur K3. Avant que l'arrêt du processus soit détecté par le leader (K1), démarrer la procédure d'adhésion sur K2. Suivant la configuration réseau, K2 doit faire un test de performance avec K3, qui n'est plus joignable. Si c'est le cas, la procédure de K2 est un échec.

L'exécution de ce dernier scénario de l'adhésion offre aussi le résultat attendu. La journalisation de K2 sur la Figure 34 montre le résultat de deux tentatives d'adhésion encadrées en rouge. Le nœud K2 envoie correctement, à chaque tentative échouée, un message d'annulation (Abort). Les tentatives échouent, car le nœud K3 n'est pas accessible pour faire des tests performance. Ce phénomène se produit lorsqu'un nœud n'est pas encore détecté comme sorti du cluster et le leader ne l'a pas encore enlevé du CGraph.

```
kollap@KNode-1: ~/kollaps/binaries

Creating the controller...
Creating the controller...
Controller created.
Initialization of the controller...
Starting the controller...
This node is not a leader, searching for leader on the subnet.
[CQ REQUEST] BROADCAST
[CQ REQUEST]: Received response, forwarding it
[CLUSTER ADD]: Adding myself in the cluster. The leader is: [192.168.100.242:8081]
[CLUSTER ADD]: CGraphGet request sent to the leader
[CLUSTER ADD]: CGraph received by the leader
[CLUSTER ADD]: Completing the graph with tests...
[PERF]: Missing information between me and [192.168.100.207:8081], starting test...
[PERF]: Issue when making test. Remaining tries: [3]. Error: []: Error type: Wrapped error
Connection refused (os error 111)          Error message: .
[PERF]: Issue when making test. Remaining tries: [2]. Error: []: Error type: Wrapped error
Connection refused (os error 111)          Error message: .
[PERF]: Issue when making test. Remaining tries: [1]. Error: []: Error type: Wrapped error
Connection refused (os error 111)          Error message: .
[PERF]: Issue when making test. Remaining tries: [0]. Error: []: Error type: Wrapped error
Connection refused (os error 111)          Error message: .
[CLUSTER ADD]: Error adding myself in the cluster, sending an ABORT. Error: [PERF]: Error type: Performance Test Failed
Error message: Exceeded number of tries..

Connection refused (os error 111)
[CLUSTER ADD]: Waiting a bit and then retry
[DOCKER HELPER]: Network kollaps_ipvlan already exists, skipping...
Initialization of the controller...
[CQ RETRY]: Remaining retries: 2
[CQ REQUEST] BROADCAST
[CQ REQUEST]: Received response, forwarding it
[CLUSTER ADD]: Adding myself in the cluster. The leader is: [192.168.100.242:8081]
[CLUSTER ADD]: CGraphGet request sent to the leader
[CLUSTER ADD]: CGraph received by the leader
[CLUSTER ADD]: Completing the graph with tests...
[PERF]: Missing information between me and [192.168.100.207:8081], starting test...
[PERF]: Issue when making test. Remaining tries: [3]. Error: []: Error type: Wrapped error
Connection refused (os error 111)          Error message: .
[PERF]: Issue when making test. Remaining tries: [2]. Error: []: Error type: Wrapped error
Connection refused (os error 111)          Error message: .
[PERF]: Issue when making test. Remaining tries: [1]. Error: []: Error type: Wrapped error
Connection refused (os error 111)          Error message: .
[PERF]: Issue when making test. Remaining tries: [0]. Error: []: Error type: Wrapped error
Connection refused (os error 111)          Error message: .
[CLUSTER ADD]: Error adding myself in the cluster, sending an ABORT. Error: [PERF]: Error type: Performance Test Failed
Error message: Exceeded number of tries..
[]: Error type: Wrapped error      Error message: .

[CLUSTER ADD]: Waiting a bit and then retry

[CQ REQUEST] BROADCAST
[CQ REQUEST]: Received response, forwarding it
[CLUSTER ADD]: Adding myself in the cluster. The leader is: [192.168.100.242:8081]
[CLUSTER ADD]: CGraphGet request sent to the leader
[CLUSTER ADD]: CGraph received by the leader
[CLUSTER ADD]: Completing the graph with tests...
[PERF]: Missing information between me and [192.168.100.242:8081], starting test...
[PERF]: Test finished
[PERF]: me -> [192.168.100.242:8081] = 64319267
[CLUSTER ADD]: Completion of the graph finished
```

Figure 34: Journalisation de K2 lors de la tentative d'adhésion avec K3 manquant.

La Figure 35 montre ce qui se passe en parallèle sur le leader (K1). Encadrées en rouge, ce sont les acceptations des deux tentatives échouées de K2. Le leader est en train de détecter la sortie de K3 du cluster (en vert). Une fois détectée, il retire K3 du CGraph et c'est à ce moment-là que K2, lors de sa requête suivante (acceptation encadrée en bleu), est capable de finaliser une procédure.

```

kollap@KNode-1: ~/kollaps/binaries
[ CQ REQUEST]: Accept request
[ TCPListener]: Connection closed by the peer: 192.168.100.200:33989
[ TCPListener]: Connection closed by the peer: 192.168.100.200:40625
[ HEARTBEAT CHECK]: Missing nodes: [ClusterNodeInfo { ip_addr: 192.168.100.207, port: 8081 }]
[ HEARTBEAT CHECK]: NodeFailure : [192.168.100.207:8081]
[ HEARTBEAT CHECK]: Missing nodes: [ClusterNodeInfo { ip_addr: 192.168.100.207, port: 8081 }]
[ HEARTBEAT CHECK]: NodeFailure : [192.168.100.207:8081]
[ CQ REQUEST]: Accept request
[ TCPListener]: Connection closed by the peer: 192.168.100.200:37293
[ TCPListener]: Connection closed by the peer: 192.168.100.200:43943
[ HEARTBEAT CHECK]: Missing nodes: [ClusterNodeInfo { ip_addr: 192.168.100.207, port: 8081 }]
[ HEARTBEAT CHECK]: NodeFailure : [192.168.100.207:8081]
[ CQ REQUEST]: Accept request
[ PERF]: Received perf request starting the server...
[ PERF]: Server started
-----
Server listening on 5201
-----
Accepted connection from 192.168.100.200, port 34622
[ 5] local 192.168.100.242 port 5201 connected to 192.168.100.200 port 34636
[ ID] Interval Transfer Bitrate
[ 5] 0.00-1.00 sec 6.35 GBytes 54.6 Gbits/sec
[ 5] 1.00-2.00 sec 7.08 GBytes 60.9 Gbits/sec
[ 5] 2.00-3.00 sec 8.67 GBytes 74.5 Gbits/sec
[ 5] 3.00-4.00 sec 8.82 GBytes 75.7 Gbits/sec
[ 5] 4.00-5.00 sec 8.70 GBytes 74.8 Gbits/sec
[ 5] 5.00-6.00 sec 8.41 GBytes 72.2 Gbits/sec
[ 5] 6.00-7.00 sec 8.51 GBytes 73.1 Gbits/sec
[ 5] 7.00-8.00 sec 7.35 GBytes 63.1 Gbits/sec
[ 5] 8.00-9.00 sec 7.72 GBytes 66.3 Gbits/sec
[ 5] 9.00-10.00 sec 7.93 GBytes 68.1 Gbits/sec
[ 5] 10.00-11.00 sec 7.18 GBytes 61.7 Gbits/sec
[ 5] 11.00-12.00 sec 7.06 GBytes 60.6 Gbits/sec
[ 5] 12.00-13.00 sec 6.34 GBytes 54.4 Gbits/sec
[ 5] 13.00-14.00 sec 7.40 GBytes 63.5 Gbits/sec
[ 5] 14.00-15.00 sec 7.52 GBytes 64.6 Gbits/sec
[ 5] 15.00-15.04 sec 287 MBytes 62.7 Gbits/sec
-----
[ ID] Interval Transfer Bitrate
[ 5] 0.00-15.04 sec 115 GBytes 65.9 Gbits/sec receiver
[ TCPListener]: Connection closed by the peer: 192.168.100.200:45841
[ TCPListener]: Connection closed by the peer: 192.168.100.200:35103
[ TCPListener]: Connection closed by the peer: 192.168.100.200:37327
[ Orchestrator] : Updating with CGraph
[ Orchestrator] : Old Residual Graph Size : 1
[ Orchestrator] : To Remove : []
[ Orchestrator] : To Add : [ClusterNodeInfo { ip_addr: 192.168.100.200, port: 8082 }]
[ Orchestrator] : UUID Affected by removes : []
[ Orchestrator] : New Residual Graph Size : 2
[ HEARTBEAT CHECK]: Cannot send receive heartbeat, it must have been aborted

```

Figure 35: Journalisation de K1 lors de la tentative d'adhésion avec K3 manquant.

La deuxième sous-catégorie de la gestion de cluster est la détection de retrait d'un nœud et/ou de son redémarrage.

6.2.2 Retrait et redémarrage

Scénario 4

Objectif : Le leader doit s'occuper de vérifier que tous les nœuds formant le cluster soient toujours présents et disponibles. Pour ce faire, il utilise un système de *heartbeat*. Le nombre de pulsations qu'un nœud non-leader peut ignorer est configurable. Dans ce test, il s'agit de 3. Le leader doit être capable de détecter la déconnexion de plusieurs nœuds en simultané.

Scénario : Créer un cluster stable formé de K1 (leader), K2 et K3. Arrêter le processus Kollaps sur K3, attendre 5 secondes et arrêter K2.

L'exécution du scénario montre que le leader est capable de détecter l'absence de plusieurs

nœuds en simultané. La Figure 36 montre la journalisation du leader lors du test. Le leader détecte d'abord l'absence de K3, puis K2 à partir de la pulsation suivante. Dans la dernière pulsation montre uniquement l'absence de K2, car K3 a déjà subit trois manquements et est donc déjà considéré comme hors du cluster.

```

[Orchestrator] : Updating with CGraph
[Orchestrator] : Old Residual Graph Size : 2
[Orchestrator] : To Remove : []
[Orchestrator] : To Add : [ClusterNodeInfo { ip_addr: 192.168.100.200, port: 8082 }]
[Orchestrator] : UUID Affected by removes : []
[Orchestrator] : New Residual Graph Size : 3
[HEARTBEAT CHECK]: Missing nodes: {ClusterNodeInfo { ip_addr: 192.168.100.207, port: 8081 }}
[HEARTBEAT]: NodeFailure : [192.168.100.207:8081]
[HEARTBEAT CHECK]: Missing nodes: {ClusterNodeInfo { ip_addr: 192.168.100.200, port: 8081 }, ClusterNodeInfo { ip_addr: 192.168.100.207, port: 8081 }}
[HEARTBEAT]: NodeFailure : [192.168.100.200:8081]
[HEARTBEAT CHECK]: Missing nodes: {ClusterNodeInfo { ip_addr: 192.168.100.207, port: 8081 }, ClusterNodeInfo { ip_addr: 192.168.100.200, port: 8081 }}
[HEARTBEAT]: NodeFailure : [192.168.100.207:8081]
[HEARTBEAT CHECK]: Missing nodes: {ClusterNodeInfo { ip_addr: 192.168.100.200, port: 8081 }}
[HEARTBEAT]: NodeFailure : [192.168.100.200:8081]

```

Figure 36: Journalisation de K1 lors de la déconnexion de K2 et K3.

Scénario 5

Objectif : Il se peut qu'un nœud présent dans le cluster subisse un redémarrage ou que l'application Kollaps est relancée. Si un redémarrage intervient, il n'y a pas besoin de refaire les tests de performance.

Scénario : Créer un cluster stable formé de K1 (leader), K2 et K3. Redémarrer le processus Kollaps sur K3.

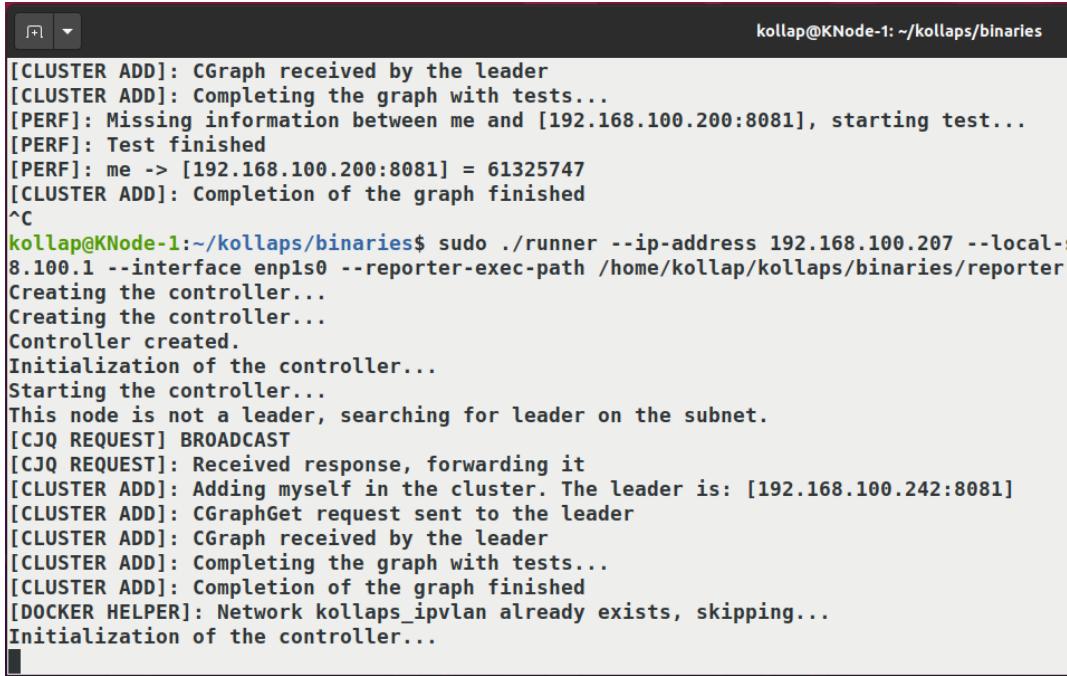
L'exécution du scénario montre que le redémarrage est bien détecté par le leader et que le nœud K3 n'a pas besoin de refaire les tests de performance. La Figure 37 montre que le nœud K1 a détecté l'absence du nœud K3 avant d'accepter sa nouvelle requête d'adhésion. Aussi, la sortie de K1 montre que lors de l'acceptation de K3 dans le cluster, l'orchestrateur n'as pas eu besoin de mettre à jour son nombre de nœuds disponibles. La Figure 38 montre que K3 n'a pas besoin d'effectuer des tests de performance lors de son redémarrage.

```

[Orchestrator] : UUID Affected by removes : []
[Orchestrator] : New Residual Graph Size : 3
[HEARTBEAT CHECK]: Missing nodes: {ClusterNodeInfo { ip_addr: 192.168.100.207, port: 8081 }}
[HEARTBEAT]: Nouvel arrivage : [192.168.100.207:8081]
[CJO REQUEST]: Accept request
[CJO REQUEST]: Detecting reboot of node already in cluster
[TCPLISTENER]: Connection closed by the peer: 192.168.100.207:35161
[TCPLISTENER]: Connection closed by the peer: 192.168.100.207:39361
[Orchestrator] : Updating with CGraph
[Orchestrator] : Old Residual Graph Size : 3
[Orchestrator] : To Remove : []
[Orchestrator] : To Add : []
[Orchestrator] : UUID Affected by removes : []
[Orchestrator] : New Residual Graph Size : 3

```

Figure 37: Journalisation de K1 lors du redémarrage de K3.



```

[CLUSTER ADD]: CGraph received by the leader
[CLUSTER ADD]: Completing the graph with tests...
[PERF]: Missing information between me and [192.168.100.200:8081], starting test...
[PERF]: Test finished
[PERF]: me -> [192.168.100.200:8081] = 61325747
[CLUSTER ADD]: Completion of the graph finished
^C
kollap@KNode-1:~/kollaps/binaries$ sudo ./runner --ip-address 192.168.100.207 --local-ip 192.168.100.1 --interface enp1s0 --reporter-exec-path /home/kollap/kollaps/binaries/reporter
Creating the controller...
Creating the controller...
Controller created.
Initialization of the controller...
Starting the controller...
This node is not a leader, searching for leader on the subnet.
[CJQ REQUEST] BROADCAST
[CJQ REQUEST]: Received response, forwarding it
[CLUSTER ADD]: Adding myself in the cluster. The leader is: [192.168.100.242:8081]
[CLUSTER ADD]: CGraphGet request sent to the leader
[CLUSTER ADD]: CGraph received by the leader
[CLUSTER ADD]: Completing the graph with tests...
[CLUSTER ADD]: Completion of the graph finished
[DOCKER HELPER]: Network kollaps_ipvlan already exists, skipping...
Initialization of the controller...

```

Figure 38: Journalisation de K3 lors de son redémarrage.

Le test de ce dernier scénario signe la fin des tests d'intégration pour la gestion du cluster. La prochaine catégorie est le déploiement et l'exécution d'une expérimentation.

6.3 Déploiement et exécution

Les scénarios attribués à cette catégorie ont pour but de montrer qu'il est possible de déployer une expérimentation sur le cluster Kollaps. Ils montrent que l'orchestrateur trouve, en fonction de la taille du cluster, différentes façons de déployer une expérimentation. Le premier scénario montre tout d'abord le bon fonctionnement d'une expérimentation.

Scénario 6

Objectif : Démontrer qu'il est possible de déployer correctement une expérimentation sur un seul nœud. Aussi, à la terminaison de cette expérimentation, les éléments déployés doivent être supprimés et nettoyés.

Scénario : Créer un cluster d'un seul nœud (K1) et demander de déployer une expérimentation.

La Figure 39 ci-dessous représente la sortie de K1. Nous pouvons voir trois parties. La première, encadrée en vert, l'OManager reçoit la nouvelle expérimentation. Une fois cette expérimentation traitée, il l'envoie aux nœuds choisis par l'orchestrateur pour l'exécution. Dans ce cas, il n'y a qu'un seul nœud et c'est lui-même. L'EManager reçoit l'expérimentation et démarre les trois processus terminaux avec leur Reporter (encadrée en bleu). Finalement, la dernière partie est l'exécution de l'expérimentation (encadrée en rouge). Les erreurs présentes dans l'exécution sont normales, elles ne proviennent pas de l'implémentation, mais du TCAL. Ce sont des erreurs bénignes. L'équipe Kollaps a ces erreurs depuis plusieurs années.

L'exécution du scénario est un succès.

```

kollap@KNode-1: ~/kollaps/binaries$ sudo ./runner --ip-address 192.168.100.242 --local-speed 100000000 --subnet 192.168.100.0/24 --gateway 192.168.100.1 --interface enp1s0 --leader --reporter-exec-path /home/kollap/kollaps/binaries/reporter
Creating the controller...
Creating the controller...
Controller created.
Initialization of the controller...
Starting the controller...
[CREATE CLUSTER]: Creating a cluster
[DOCKER HELPER]: Network kollaps_ipvlan already exists, skipping...
Initialization of the controller...
[OManager] : New Topology
[OManager] : Sending new Emulation to these nodes: [ClusterNodeInfo { ip_addr: 192.168.100.242, port: 8082 }]
[OManager] : Topology Accepted
[EManager] : Received a new experiment from local node
[EMULCORE 0e8a61f9-5f53-4e6e-ba88-d9ffe0dd2fc6]: Starting reporter for app: 0 : c1
[REPORTER: 192.168.100.100] is ready, listening for flows changes
[EMULCORE 0e8a61f9-5f53-4e6e-ba88-d9ffe0dd2fc6]: Starting reporter for app: 1 : c2
[REPORTER: 192.168.100.101] is ready, listening for flows changes
[EMULCORE 0e8a61f9-5f53-4e6e-ba88-d9ffe0dd2fc6]: Starting reporter for app: 2 : c3
[REPORTER: 192.168.100.102] is ready, listening for flows changes
[EManager] : Emulcore 0e8a61f9-5f53-4e6e-ba88-d9ffe0dd2fc6 is ready. Sending Ready to OManager: [192.168.100.242:8080]
[OManager] : Emulation 0e8a61f9-5f53-4e6e-ba88-d9ffe0dd2fc6 is ready, sending start
[TCPListener]: Connection closed by the peer: 192.168.100.242:43167
[EManager] Starting synchre for 0e8a61f9-5f53-4e6e-ba88-d9ffe0dd2fc6
RTNETLINK answers: Invalid argument
RTNETLINK answers: Invalid argument
We have an error talking to the kernel
We have an error talking to the kernel
RTNETLINK answers: Invalid argument
We have an error talking to the kernel
RTNETLINK answers: No such file or directory
We have an error talking to the kernel
RTNETLINK answers: No such file or directory
We have an error talking to the kernel
filter parent 1: protocol ip pref 1 u32 chain 0 fh 800: ht divisor 1
filter parent 1: protocol ip pref 1 u32 chain 0 fh 800: ht divisor 1
RTNETLINK answers: No such file or directory
We have an error talking to the kernel
filter parent 4: protocol ip pref 2 u32 chain 0 fh f64: ht divisor 256
filter parent 4: protocol ip pref 2 u32 chain 0 fh f64: ht divisor 256
filter parent 1: protocol ip pref 1 u32 chain 0 fh 800: ht divisor 1
filter parent 4: protocol ip pref 2 u32 chain 0 fh f64: ht divisor 256

```

Figure 39: Journalisation de K1 lors du déploiement de la topologie sur un seul nœud.

Dans la deuxième image (Figure 40), nous voyons les différents conteneurs se faire supprimer après l’expérimentation.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
00a1385486fe	kollaps-ipperf1.0	"/bin/sh"	About a minute ago	Up About a minute		kollaps_0e8a61f9-5f53-4e6e-ba88-d9ffe0dd2fc6_2
ce5f6c6f800e	kollaps-ipperf1.0	"/bin/sh"	About a minute ago	Up About a minute		kollaps_0e8a61f9-5f53-4e6e-ba88-d9ffe0dd2fc6_1
c660c8399392	kollaps-ipperf1.0	"/bin/sh"	About a minute ago	Up About a minute		kollaps_0e8a61f9-5f53-4e6e-ba88-d9ffe0dd2fc6_0
00a1385486fe	kollaps-ipperf1.0	"/bin/sh"	4 minutes ago	Up 4 minutes		kollaps_0e8a61f9-5f53-4e6e-ba88-d9ffe0dd2fc6_2
ce5f6c6f800e	kollaps-ipperf1.0	"/bin/sh"	4 minutes ago	Up 4 minutes		kollaps_0e8a61f9-5f53-4e6e-ba88-d9ffe0dd2fc6_1
c660c8399392	kollaps-ipperf1.0	"/bin/sh"	4 minutes ago	Up 4 minutes		kollaps_0e8a61f9-5f53-4e6e-ba88-d9ffe0dd2fc6_0
00a1385486fe	kollaps-ipperf1.0	"/bin/sh"	5 minutes ago	Up 5 minutes		kollaps_0e8a61f9-5f53-4e6e-ba88-d9ffe0dd2fc6_2
ce5f6c6f800e	kollaps-ipperf1.0	"/bin/sh"	5 minutes ago	Up 5 minutes		kollaps_0e8a61f9-5f53-4e6e-ba88-d9ffe0dd2fc6_1
c660c8399392	kollaps-ipperf1.0	"/bin/sh"	5 minutes ago	Up 5 minutes		kollaps_0e8a61f9-5f53-4e6e-ba88-d9ffe0dd2fc6_0

Figure 40: Aperçu des conteneurs Docker au fil de l’expérimentation sur K1.

Scénario 7

Objectif : Démontrer qu’il est possible de déployer correctement une expérimentation sur plusieurs noeuds et que les différents EmulCores répartis sur les nœuds communiquent correctement pour émuler l’expérimentation.

Scénario : Créer un cluster de trois nœuds : K1 (leader), K2 et K3. Déployer une expérimentation de trois processus terminaux et exécuter des tests de performance iPerf3 pour valider l’expérimentation.

L'exécution du scénario se déroule sans problème et produit le résultat attendu.

Les trois Figures suivantes (Figures 41 pour K1, 42 pour K2, 43 pour K3) montrent la sortie de chacune des machines. Tout d'abord, l'OManager reçoit la topologie à déployer et demande à l'orchestrateur de trouver un déploiement. Une fois l'expérimentation acceptée, l'OManager envoie l'expérimentation à tous les nœuds concernés (premier encadré rouge sur K1). Là, les EManagers de chaque machine créent un EmulCore qui déploie les processus de l'expérimentation. À la fin du déploiement, nous voyons qu'ils avertissent l'OManager qu'ils sont prêts. Cette procédure est encadrée en vert sur les trois captures. Une fois l'expérimentation prête partout, l'OManager envoie le signal de départ (deuxième encadré rouge sur la sortie de K1). Là, démarre le processus de synchronisation entre les EmulCores (encadré en bleu sur les trois captures). Comme il l'est indiqué sur la capture de K2, c'est son EmulCore qui est le leader et qui s'occupe de la synchronisation. Ensuite, les encadrés oranges montrent le partage de messages entre EmulCores permettant de s'avertir des changements de flux.

```

kollap@KNode-1: ~/kollaps/binaries
kollap@KNode-1: ~/kollaps/binaries
kollap@KNode-1: ~/kollaps/binaries

[OManager] : New Topology
[OManager] : Sending new Emulation to these nodes: [ClusterNodeInfo { ip_addr: 192.168.100.200, port: 8082 }, ClusterNodeInfo { ip_addr: 192.168.100.242, port: 8082 }, ClusterNodeInfo { ip_addr: 192.168.100.207, port: 8082 }]
[OManager] : Sending new Emulation to [192.168.100.200:8082]
[OManager] : Sending new Emulation to [192.168.100.207:8082]

[EManager] : Received a new experiment from local node
[OManager] : Topology Accepted
[EMULCORE 7fba6fea-5c75-44eb-89e1-09ea0126cbf4]: Starting reporter for app: 1 : c2
[TCPListener]: Connection closed by the peer: 192.168.100.200:37767
[REPORTER: 192.168.100.101] is ready, listening for flows changes
[EManager] : EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4 is ready. Sending Ready to OManager: [192.168.100.242:8082]
[TCPListener]: Connection closed by the peer: 192.168.100.242:35116
[OManager] : Emulation 7fba6fea-5c75-44eb-89e1-09ea0126cbf4 is ready, sending start

[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : Synchronization my next host: [192.168.100.200:9090]
[TCPListener]: Connection closed by the peer: 192.168.100.207:38325
[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : Syncro : Received begin time, passing to next: Instant { tv_sec: 15286, tv_nsec: 568948366 }
[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : Syncro : Confirmation Ok, passing to next
[TCPListener]: Stop listening for new TCP connections: Ok("TCPListener": Closing Socket")
[TCPListener]: Stop reading on TCPStream: Ok("Stop listening on TCP Stream")

[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : We have an error talking to the kernel
[TNETLINK answers: No such file or directory]
[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : We have an error talking to the kernel
[filter parent 1: protocol ip pref 1 u32 chain 0 fh 800: ht divisor 1]
[filter parent 4: protocol ip pref 2 u32 chain 0 fh f64: ht divisor 256]
[Flow updated between [APP c1] IP 192.168.100.100 Host [192.168.100.207:8082] and [APP c3] IP 192.168.100.102 Host [192.168.100.200:8082] : Some(20254)]
[TCPListener]: Connection closed by the peer: 192.168.100.207:41583
[Flow updated between [APP c3] IP 192.168.100.102 Host [192.168.100.200:8082] and [APP c1] IP 192.168.100.100 Host [192.168.100.207:8082] : Some(229)]
[TCPListener]: Connection closed by the peer: 192.168.100.200:44073

```

Figure 41: Capture de K1 lors d'une expérimentation distribuée.

```

kollap@KNode-1: ~/kollaps/binaries
kollap@KNode-1: ~/kollaps/binaries
kollap@KNode-1: ~/kollaps/binaries

[EManager] : Received a new experiment from local node
[TCPListener]: Connection closed by the peer: 192.168.100.242:45293
[EMULCORE 7fba6fea-5c75-44eb-89e1-09ea0126cbf4]: Starting reporter for app: 2 : c3
[REPORTER: 192.168.100.102] is ready, listening for flows changes
[EManager] : EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4 is ready. Sending Ready to OManager: [192.168.100.242:8080]
[EManager] : Starting syncro for 7fba6fea-5c75-44eb-89e1-09ea0126cbf4

[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : Synchronization, my next host: [192.168.100.207:9090]
[TCPListener]: Connection closed by the peer: 192.168.100.242:36133
[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : I am the Leader of the emulation, start Syncro
[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : Syncro : first proposed time: Instant { tv_sec: 9552, tv_nsec: 361139295 }
[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : Syncro : first round in: 177.627us
[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : Syncro : enough time for validation
[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : Syncro : Leader finish syncro, OK
[TCPListener]: Connection closed by the peer: 192.168.100.242:40375
[TCPListener]: Stop Listening for new TCP connections: Ok("TCPListener": Closing Socket")
[read -c0]: run-time worker panicked at called `Result::unwrap()` on an `Err` value: "Stop listening on TCP Stream", /home/kollap/kollaps/tcp_listener.rs:207:68
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
[TNETLINK answers: Invalid argument]
[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : We have an error talking to the kernel
[TNETLINK answers: No such file or directory]
[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : We have an error talking to the kernel
[filter parent 1: protocol ip pref 1 u32 chain 0 fh 800: ht divisor 1]
[filter parent 4: protocol ip pref 2 u32 chain 0 fh f64: ht divisor 256]
[Flow updated between [APP c1] IP 192.168.100.100 Host [192.168.100.207:8082] and [APP c3] IP 192.168.100.102 Host [192.168.100.200:8082] : Some(20254)]
[TCPListener]: Connection closed by the peer: 192.168.100.207:39883

```

Figure 42: Capture de K2 lors d'une expérimentation distribuée.

```

[kollap@KNode-1: ~/kollaps/binaries]
[EManager] : Received a new experiment from local node
[TCPListener]: Connection closed by the peer: 192.168.100.242:35243
[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4]: Starting reporter for app: 0 : c1
[REPORTER: 192.168.100.100] is ready, listening for flow changes
[EManager] : EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4 is ready. Sending Ready to OManager: [192.168.100.242:8080]
[EManager] Starting synchro for 7fba6fea-5c75-44eb-89e1-09ea0126cbf4
[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : Synchronization, my next host: [192.168.100.242:9090]
[TCPListener]: Connection closed by the peer: 192.168.100.242:40813
[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : Syncro : Received begin time, passing to next: Instant { tv_sec: 15218, tv_nsec: 604230390 }
[EmulCore 7fba6fea-5c75-44eb-89e1-09ea0126cbf4] : Syncro : Confirmation Ok, passing to next
[TCPListener] Stop listening for new TCP connections: Ok("[TCPListener]: Closing Socket")
[TCPListener]: Stop reading on TCPStream: Ok("Stop listening on TCP Stream")

KINETLINK answers: invalid argument
We have an error talking to the kernel
KINETLINK answers: No such file or directory
We have an error talking to the kernel
filter parent 1: protocol ip pref 1 u32 chain 0 fh 800: ht divisor 1
filter parent 4: protocol ip pref 2 u32 chain 0 fh f64: ht divisor 256
Flow updated between [APP c1] IP 192.168.100.100 Host [192.168.100.207:8082] and [APP c3] IP 192.168.100.102 Host [192.168.100.200:80]
[2] : Some(20254)
[EmulCore] BW UPDATE : c1 -> c3 : 102400
Flow updated between [APP c3] IP 192.168.100.102 Host [192.168.100.200:8082] and [APP c1] IP 192.168.100.100 Host [192.168.100.207:80]
[2] : Some(229)
[TCPListener]: Connection closed by the peer: 192.168.100.200:39903
[EmulCore] BW UPDATE : c1 -> c3 : 102400
Flow updated between [APP c1] IP 192.168.100.100 Host [192.168.100.207:8082] and [APP c3] IP 192.168.100.102 Host [192.168.100.200:80]
[2] : Some(96397)
[EmulCore] BW UPDATE : c1 -> c3 : 102400
Flow updated between [APP c3] IP 192.168.100.102 Host [192.168.100.200:8082] and [APP c1] IP 192.168.100.100 Host [192.168.100.207:80]
[2] : Some(189)
[TCPListener]: Connection closed by the peer: 192.168.100.200:37663
[EmulCore] BW UPDATE : c1 -> c3 : 102400

```

Figure 43: Capture de K3 lors d'une expérimentation distribuée.

Une fois le déploiement distribué d'une expérimentation validé, il faut tester que le partage de bande passante et les propriétés réseau indiquées sur l'expérimentation sont respectées par l'émulation.

Pour ce faire, le processus C3 (déployé sur K2) sert de serveur iPerf3. C3 exécute deux instances du serveur iPerf3 en simultané. La Figure 44 montre les deux instances, la première écoute sur le port par 5201 (à gauche) et la deuxième sur le port 5202 (à droite). Ensuite, C1 (déployé sur K3) démarre un test iPerf3 de longue durée en direction du port par défaut sur C3 (Figure 45). Après quelques secondes, C2 (déployé sur K1) démarre aussi son test de performance en direction de C3 mais sur le port 5202 (Figure 46). Le test lancé par C2 est plus court que le test lancé par C1, de ce fait, il commence après et se termine avant.

The screenshot shows two terminal windows side-by-side. Both are running on a host named 'kollap@KNode-1'.

Left Terminal:

```
kollap@KNode-1:~/kollaps/binaries$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
bf44afadffba kollaps-iperf:1.0 "/bin/sh" 52 seconds ago Up 51 s
(seconds)
kollaps_7ba6fea-5c75-44eb-89e1-09ea0126cbf4_2
kollap@KNode-1:~/kollaps/binaries$ docker exec -it bf44afadffba /bin/sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
7: eth0@if2: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc pr
io state UNKNOWN qlen 1000
    link/ether 52:54:00:03:09:ae brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.102/24 brd 192.168.100.255 scope global eth0
        valid_lft forever preferred_lft forever
/ # iperf3 -s
-----
```

Right Terminal:

```
kollap@KNode-1:~/kollaps/binaries$ docker exec -it bf44afadffba /bin/sh
/ # iperf3 -p 5202 -s
-----
Server listening on 5202 (test #1)
-----
Accepted connection from 192.168.100.101, port 40208
[ 5] local 192.168.100.102 port 5202 connected to 192.168.100.101 port
+ 40222
[ ID] Interval      Transfer     Bitrate
[ 5]  0.00-1.00   sec  5.54 MBytes  46.5 Mbytes/sec
[ 5]  1.00-2.00   sec  5.77 MBytes  48.4 Mbytes/sec
[ 5]  2.00-3.00   sec  5.79 MBytes  48.6 Mbytes/sec
[ 5]  3.00-4.00   sec  5.59 MBytes  46.9 Mbytes/sec
[ 5]  4.00-5.00   sec  5.70 MBytes  47.8 Mbytes/sec
[ 5]  5.00-6.00   sec  5.69 MBytes  47.7 Mbytes/sec
[ 5]  6.00-7.00   sec  5.64 MBytes  47.3 Mbytes/sec
[ 5]  7.00-8.00   sec  5.77 MBytes  48.0 Mbytes/sec
[ 5]  8.00-9.00   sec  5.75 MBytes  48.3 Mbytes/sec
[ 5]  9.00-10.00  sec  5.73 MBytes  48.1 Mbytes/sec
[ 5] 10.00-10.51  sec  2.94 MBytes  48.8 Mbytes/sec
-----
```

Bottom Terminal:

```
[ ID] Interval      Transfer     Bitrate
[ 5]  0.00-10.51  sec  59.9 MBytes  47.8 Mbytes/sec
receiver
-----
```

Bottom Right Terminal:

```
Server listening on 5202 (test #2)
kollap@KNode-1:~/kollaps/binaries$
```

Two sections of the left terminal's iPerf3 output are highlighted with red boxes:

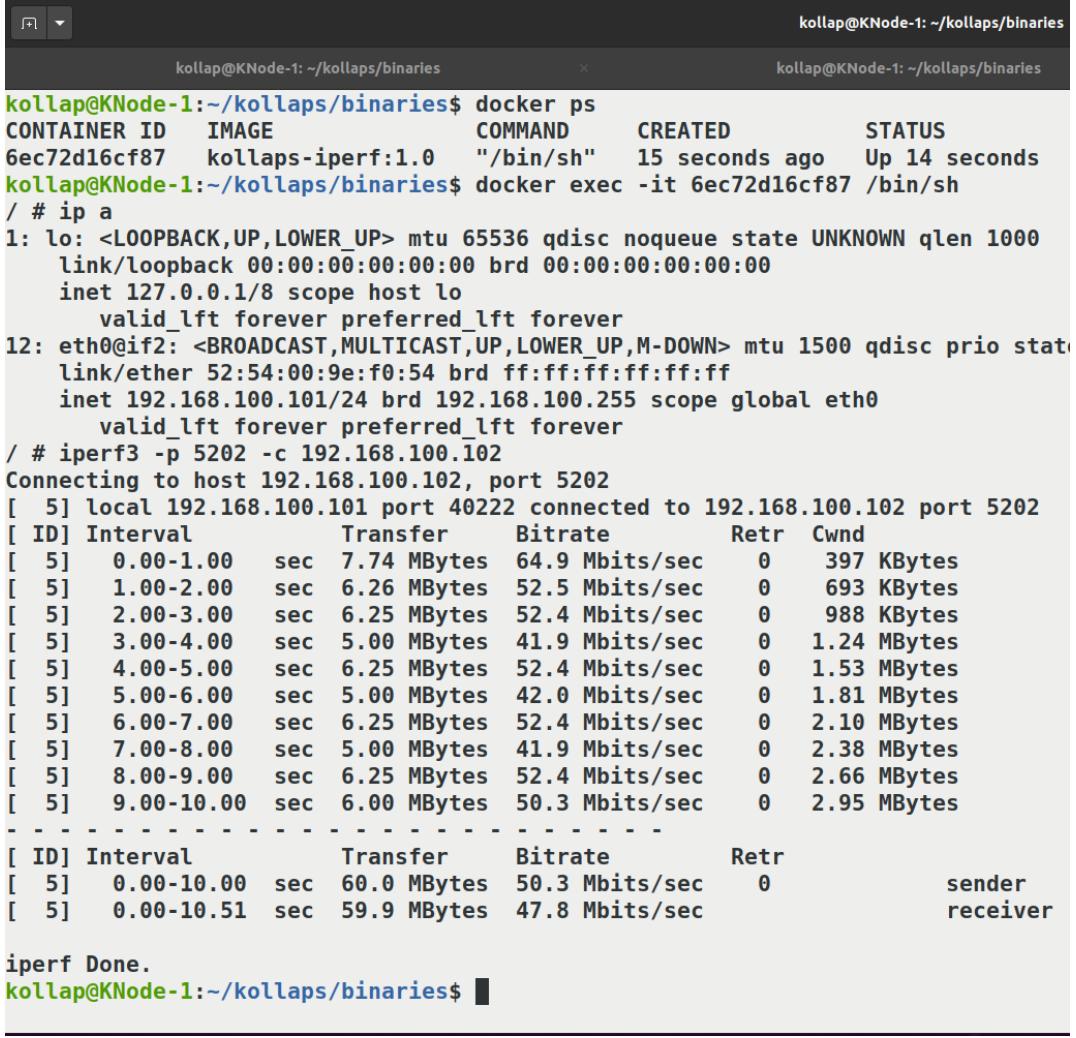
- A large box highlights the first 10 seconds of the test, showing a stable transfer rate around 47-48 Mbytes/sec.
- A smaller box highlights the last second of the test, showing a sharp drop in transfer rate to approximately 2.94 Mbytes/sec.

Figure 44: Déploiement de C3 sur K2. À gauche : serveur iPerf3 écoutant sur le port par défaut et utilisé par C1. À droite : serveur iPerf3 écoutant sur le port 5202 et utilisé par C2.

```
kollap@KNode-1: ~/kollaps/binaries$ docker ps
CONTAINER ID   IMAGE      COMMAND   CREATED      STATUS
c096ea26a126  kollaps-iperf:1.0 "/bin/sh"   About a minute ago   Up About a minute ago
26cbf4_0

kollap@KNode-1:~/kollaps/binaries$ docker exec -it c096ea26a126 /bin/sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
11: eth0@if2: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc prio state
link/ether 52:54:00:89:96:3f brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.100/24 brd 192.168.100.255 scope global eth0
        valid_lft forever preferred_lft forever
/ # iperf3 -t 100 -c 192.168.100.102
Connecting to host 192.168.100.102, port 5201
[  5] local 192.168.100.100 port 46954 connected to 192.168.100.102 port 5201
[ ID] Interval           Transfer     Bitrate      Retr  Cwnd
[  5]  0.00-1.00   sec  14.9 MBytes  125 Mbits/sec  0  1.36 MBytes
[  5]  1.00-2.00   sec  11.2 MBytes  94.4 Mbits/sec  0  1.94 MBytes
[  5]  2.00-3.00   sec  11.2 MBytes  94.4 Mbits/sec  0  2.52 MBytes
[  5]  3.00-4.00   sec  11.2 MBytes  94.4 Mbits/sec  0  3.00 MBytes
[  5]  4.00-5.00   sec  12.5 MBytes  105 Mbits/sec  0  3.00 MBytes
[  5]  5.00-6.00   sec  8.75 MBytes  73.4 Mbits/sec  0  3.00 MBytes
[  5]  6.00-7.00   sec  6.12 MBytes  51.4 Mbits/sec  0  3.00 MBytes
[  5]  7.00-8.00   sec  6.25 MBytes  52.4 Mbits/sec  0  3.00 MBytes
[  5]  8.00-9.00   sec  5.00 MBytes  41.9 Mbits/sec  0  3.00 MBytes
[  5]  9.00-10.00  sec  6.25 MBytes  52.4 Mbits/sec  0  3.00 MBytes
[  5] 10.00-11.00  sec  6.25 MBytes  52.4 Mbits/sec  0  3.00 MBytes
[  5] 11.00-12.00  sec  6.25 MBytes  52.4 Mbits/sec  0  3.00 MBytes
[  5] 12.00-13.00  sec  6.25 MBytes  52.4 Mbits/sec  0  3.00 MBytes
[  5] 13.00-14.00  sec  6.25 MBytes  52.5 Mbits/sec  0  3.00 MBytes
[  5] 14.00-15.00  sec  6.25 MBytes  52.4 Mbits/sec  0  3.00 MBytes
[  5] 15.00-16.00  sec  6.25 MBytes  52.4 Mbits/sec  0  3.00 MBytes
[  5] 16.00-17.00  sec  6.25 MBytes  52.5 Mbits/sec  0  3.00 MBytes
[  5] 17.00-18.00  sec  6.25 MBvtes  52.4 Mbits/sec  0  3.00 MBvtes
[  5] 18.00-19.00  sec  12.5 MBytes  105 Mbits/sec  0  3.00 MBytes
[  5] 19.00-20.00  sec  11.2 MBytes  94.4 Mbits/sec  0  3.00 MBytes
[  5] 20.00-21.00  sec  11.2 MBytes  94.4 Mbits/sec  0  3.00 MBytes
[  5] 21.00-22.00  sec  12.5 MBytes  105 Mbits/sec  0  3.00 MBytes
[  5] 22.00-23.00  sec  11.2 MBytes  94.4 Mbits/sec  0  3.00 MBytes
[  5] 23.00-24.00  sec  11.2 MBytes  94.4 Mbits/sec  0  3.00 MBytes
[  5] 24.00-25.00  sec  11.2 MBytes  94.4 Mbits/sec  0  3.00 MBytes
[  5] 25.00-26.00  sec  12.5 MBytes  105 Mbits/sec  0  3.00 MBytes
```

Figure 45: Déploiement de C1 sur K3.



```

kollap@KNode-1:~/kollaps/binaries$ docker ps
CONTAINER ID   IMAGE      COMMAND   CREATED      STATUS
6ec72d16cf87   kollaps-iperf:1.0   "/bin/sh"  15 seconds ago Up 14 seconds
kollap@KNode-1:~/kollaps/binaries$ docker exec -it 6ec72d16cf87 /bin/sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
        inet 127.0.0.1/8 scope host lo
            valid_lft forever preferred_lft forever
12: eth0@if2: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc prio stat
    link/ether 52:54:00:9e:f0:54 brd ff:ff:ff:ff:ff:ff
        inet 192.168.100.101/24 brd 192.168.100.255 scope global eth0
            valid_lft forever preferred_lft forever
/ # iperf3 -p 5202 -c 192.168.100.102
Connecting to host 192.168.100.102, port 5202
[ 5] local 192.168.100.101 port 40222 connected to 192.168.100.102 port 5202
[ ID] Interval      Transfer     Bitrate      Retr  Cwnd
[ 5]  0.00-1.00    sec  7.74 MBytes  64.9 Mbits/sec  0  397 KBytes
[ 5]  1.00-2.00    sec  6.26 MBytes  52.5 Mbits/sec  0  693 KBytes
[ 5]  2.00-3.00    sec  6.25 MBytes  52.4 Mbits/sec  0  988 KBytes
[ 5]  3.00-4.00    sec  5.00 MBytes  41.9 Mbits/sec  0  1.24 MBytes
[ 5]  4.00-5.00    sec  6.25 MBytes  52.4 Mbits/sec  0  1.53 MBytes
[ 5]  5.00-6.00    sec  5.00 MBytes  42.0 Mbits/sec  0  1.81 MBytes
[ 5]  6.00-7.00    sec  6.25 MBytes  52.4 Mbits/sec  0  2.10 MBytes
[ 5]  7.00-8.00    sec  5.00 MBytes  41.9 Mbits/sec  0  2.38 MBytes
[ 5]  8.00-9.00    sec  6.25 MBytes  52.4 Mbits/sec  0  2.66 MBytes
[ 5]  9.00-10.00   sec  6.00 MBytes  50.3 Mbits/sec  0  2.95 MBytes
[ -----
[ ID] Interval      Transfer     Bitrate      Retr  Cwnd
[ 5]  0.00-10.00   sec  60.0 MBytes  50.3 Mbits/sec  0
[ 5]  0.00-10.51   sec  59.9 MBytes  47.8 Mbits/sec
                                         sender
                                         receiver

iperf Done.
kollap@KNode-1:~/kollaps/binaries$ 

```

Figure 46: Déploiement de C2 sur K1.

Au début, nous voyons que C1 communique avec un débit d'environ 100 Mbit/s (encadré bleu sur la Figure 45). Lors de l'activation du test de C2, les deux flux doivent se partager le lien entre S1 et S2 et donc reçoivent tous deux 50 Mbit/s (encadrés rouges sur les Figures 44 et 45). À la fin du test de C2, C1 récupère l'entièreté de la bande passante et repasse à un débit de 100 Mbit/s. Les déploiements sont finalement nettoyés correctement à la fin de l'expérimentation.

Scénario 8

Objectif : Lorsqu'une expérimentation est distribuée, si un des nœuds supportant cette expérimentation devient inaccessible, il est alors nécessaire d'annuler l'expérimentation.

Scénario : Créer un cluster de trois nœuds : K1 (leader), K2 et K3. Déployer une expérimentation de trois processus terminaux. Durant l'expérimentation, arrêter Kollaps sur le nœud K2.

La Figure 47 montre le journal de K1. En bleu, nous voyons la partie émulation, où l'expérimentation est en cours. Ensuite, vient la partie rouge qui est la détection du CManager que le nœud K2 n'est plus accessible. Une fois son retrait du cluster confirmé, l'orchestrateur est notifié et signal à l'OManager d'annuler toutes les expérimentations qui utilisent ce nœud. Il s'agit de la partie verte. Nous voyons alors que l'OManager avertit

son propre EManager et l'EManager de K3 (Figure 48 utilisant le même code couleur).

```
[Orchestrator] : UUID Affected by removes : []
[Orchestrator] : New Residual Graph Size : 3
[OManager] : New Topology
[OManager] : Sending new Emulation to these nodes: [ClusterNodeInfo { ip_addr: 192.168.100.2
{ ip_addr: 192.168.100.207, port: 8082 }, ClusterNodeInfo { ip_addr: 192.168.100.200, port:
[OManager]: Sending new Emulation to [192.168.100.207:8082]
[EManager] : Received a new experiment from local node
[OManager]: Sending new Emulation to [192.168.100.200:8082]
[OManager] : Topology Accepted
[EMULCORE 710c1e38-5669-4e73-bd6c-af6c25426a63]: Starting reporter for app: 0 : c1
[REPORTER: 192.168.100.100] is ready, listening for flows changes
[EManager] : EmulCore 710c1e38-5669-4e73-bd6c-af6c25426a63 is ready. Sending Ready to OManager
[TCPListener]: Connection closed by the peer: 192.168.100.242:38277
[TCPListener]: Connection closed by the peer: 192.168.100.207:44509
[OManager] : Emulation 710c1e38-5669-4e73-bd6c-af6c25426a63 is ready, sending start
[EManager] Starting synchro for 710c1e38-5669-4e73-bd6c-af6c25426a63
[EmulCore 710c1e38-5669-4e73-bd6c-af6c25426a63] : Synchronization, my next host: [192.168.10
[TCPListener]: Connection closed by the peer: 192.168.100.200:40267
[EmulCore 710c1e38-5669-4e73-bd6c-af6c25426a63] : Synchro : Received begin time, passing to
nsec: 936948397 }
[EmulCore 710c1e38-5669-4e73-bd6c-af6c25426a63] : Synchro : Confirmation Ok, passing to next
[TCPListener] Stop listening for new TCP connections: Ok("[TCPListener]: Closing Socket")
[TCPListener]: Stop reading on TCPStream: Ok("Stop listening on TCP Stream")
RTNETLINK answers: Invalid argument
We have an error talking to the kernel
RTNETLINK answers: No such file or directory
We have an error talking to the kernel
filter parent 1: protocol ip pref 1 u32 chain 0 fh 800: ht divisor 1
filter parent 4: protocol ip pref 2 u32 chain 0 fh fc4: ht divisor 256
[HEARTBEAT CHECK]: Missing nodes: {ClusterNodeInfo { ip_addr: 192.168.100.200, port: 8081 }}
[HEARTBEAT]: NodeFailure : [192.168.100.200:8081]
[HEARTBEAT CHECK]: Missing nodes: {ClusterNodeInfo { ip_addr: 192.168.100.200, port: 8081 }}
[HEARTBEAT]: NodeFailure : [192.168.100.200:8081]
[HEARTBEAT CHECK]: Missing nodes: {ClusterNodeInfo { ip_addr: 192.168.100.200, port: 8081 }}
[HEARTBEAT]: NodeFailure : [192.168.100.200:8081]
[EManager] Emulation aborted 710c1e38-5669-4e73-bd6c-af6c25426a63
[NetHelper]: Stop listening for new Unix connections: Ok("stopping accept loop")
[NetHelper]: Stop reading on UnixStream: Ok("Stop listening on Unix Stream")
[WARNING]: Wanted to abort emulation 710c1e38-5669-4e73-bd6c-af6c25426a63 on node [192.168.10
[REPORTER: 192.168.100.100] - Stopping and cleaning...
[NetHelper]: Stop listening for new Unix connections: Ok("stopping accept loop")
[NetHelper]: Stop reading on UnixStream: Ok("Stop listening on Unix Stream")
```

Figure 47: Journalisation de K1 lors de l'arrêt d'un nœud supportant une expérimentation distribuée.

```
[PERF]: test finished
[PERF]: me -> [192.168.100.242:8081] = 65149824
[CLUSTER ADD]: Completion of the graph finished
[EManager]: Received a new experiment from local node
[TCPListener]: Connection closed by the peer: 192.168.100.242:39993
[EMULCORE 710c1e38-5669-4e73-bd6c-af6c25426a63]: Starting reporter for app: 2 : c3
[REPORTER: 192.168.100.102] is ready, listening for flows changes
[EManager]: EmulCore 710c1e38-5669-4e73-bd6c-af6c25426a63 is ready. Sending Ready to
[EManager] Starting synchro for 710c1e38-5669-4e73-bd6c-af6c25426a63
[EmulCore 710c1e38-5669-4e73-bd6c-af6c25426a63] : Synchronization, my next host: [192.168.100.242]
[TCPListener]: Connection closed by the peer: 192.168.100.242:35207
[EmulCore 710c1e38-5669-4e73-bd6c-af6c25426a63] : Synchro : Received begin time, passing
[EmulCore 710c1e38-5669-4e73-bd6c-af6c25426a63] : Synchro : Confirmation Ok, passing
[TCPListener]: Connection closed by the peer: 192.168.100.200:41669
[TCPListener] Stop listening for new TCP connections: Ok("[TCPListener]: Closing Socket")
[Thread 'tokio-runtime-worker' panicked at 'called `Result::unwrap()` on an `Err` value'
  file: "src/tcp.rs", line: 207, column: 68
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
[TNETLINK answers: Invalid argument
We have an error talking to the kernel
[TNETLINK answers: No such file or directory
We have an error talking to the kernel
filter parent 1: protocol ip pref 1 u32 chain 0 fh 800: ht divisor 1
filter parent 4: protocol ip pref 2 u32 chain 0 fh f64: ht divisor 256
[EManager] Emulation aborted 710c1e38-5669-4e73-bd6c-af6c25426a63
[TCPListener]: Connection closed by the peer: 192.168.100.242:43955
[NetHelper]: Stop listening for new Unix connections: Ok("stopping accept loop")
[NetHelper]: Stop reading on UnixStream: Ok("Stop listening on Unix Stream")
[REPORTER: 192.168.100.102] - Stopping and cleaning...
[NetHelper]: Stop listening for new Unix connections: Ok("stopping accept loop")
[NetHelper]: Stop reading on UnixStream: Ok("Stop listening on Unix Stream")
```

Figure 48: Journalisation de K3 lors de l’arrêt d’un nœud supportant une expérimentation distribuée.

L’exécution du scénario est un succès.

Ce dernier scénario conclut le chapitre des tests. Ces tests montrent les fonctionnalités principales amenées par ce projet. Ils montrent aussi que les fonctionnalités de bases sont toujours présentes après une réécriture complète du logiciel. Un logiciel qui fonctionne ne veut pas dire un logiciel terminé. Le chapitre suivant fait un point sur l’état actuel du projet et des améliorations nécessaires afin de devenir une plateforme stable est utilisable à grande échelle.

7 Limitations et améliorations

Le logiciel produit est une preuve de faisabilité. Il montre qu’il est possible de construire un environnement distribué supportant l’ajout et le retrait dynamique de machines.

Il a été développé dans l’esprit de pouvoir facilement intégrer toutes les fonctionnalités qu’offre actuellement Kollaps, sans pour autant toutes les implémenter. Typiquement, cette nouvelle version supporte uniquement les expérimentations avec des processus terminaux gérés par Kollaps, et pas la modification de processus déjà existants. Aussi, il ne supporte pas actuellement les processus terminaux en mode *baremetal*. L’implémentation prévoit ces fonctionnalités et permet de les ajouter facilement.

Une autre fonctionnalité qui n’a pas été portée sur cette version est le support de tous les événements d’expérimentation Kollaps. Certains événements ne sont pas disponibles, mais le code est prêt à recevoir ces fonctionnalités.

En ce qui concerne l’orchestrateur actuel, il permet de répartir la charge entre les nœuds

d'un cluster, basé sur une charge maximale autorisée par nœud. Ce système précaire permet aujourd'hui d'empêcher l'orchestrateur de distribuer toutes les expérimentations sur les mêmes nœuds. La Figure 49 ci-dessous montre le nombre de PTs déployés (couleur) sur chacun des nœuds Kollaps (10 nœuds, axe des abscisses) après des ajouts successifs (13 ajouts successifs, axe des ordonnées) d'une même expérimentation comportant trois PTs. L'axe des ordonnées montre les ajouts séquentiels, et donc montre aussi l'évolution dans le temps. Dans le cas de ce test, le nombre maximal de PTs que peut supporter un nœud est fixé au nombre de quatre.

Le graphique montre que les 4 premiers ajouts utilisent les trois premiers nœuds, puis, les quatre suivants utilisent les nœuds 3 à 5, jusqu'à arriver au stade où neuf nœuds sont pleins. Finalement, le dernier déploiement possible consiste à mettre les trois PTs de l'expérimentation sur le nœud numéro 9.

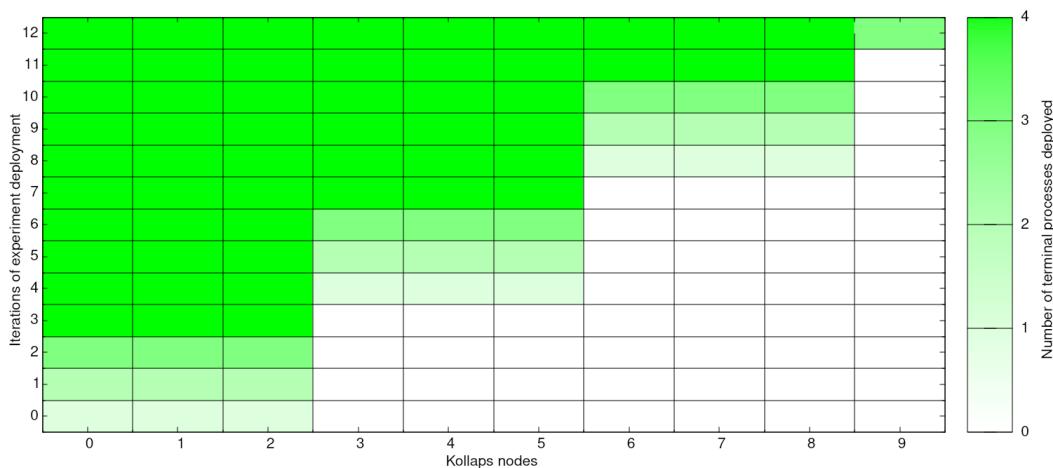


Figure 49: Distribution d'expérimentations de trois processus terminaux.

Afin d'améliorer l'orchestrateur sur la répartition de charge, plusieurs techniques sont possibles. Une première technique serait de sélectionner en premier les nœuds ayant peu de charge. Aussi, avec l'implémentation actuelle, il serait possible de répartir aléatoirement la charge en produisant des combinaisons (lors de la recherche d'isomorphisme de graphe) aléatoires, et non séquentielles. Actuellement, c'est à cause de la génération de combinaisons séquentielles que l'orchestrateur essaie toujours de déployer des expérimentations en commençant par tester les mêmes nœuds.

Un autre axe d'amélioration pour ce projet est l'implémentation d'un protocole Kollaps (voir Ch. 3.11) qui permettrait de définir un réseau virtuel dédié à Kollaps pour interconnecter les PTs d'une expérimentation.

Finalement, le code aurait aussi besoin d'être retravaillé à certains endroits. Surtout sur les premières fonctionnalités implémentées. Le Rust est un nouveau langage pour l'auteur, qui a pris du temps à un maîtrisé, ce qui implique que les premières lignes ne suivent pas forcément les bonnes pratiques de Rust et peuvent être améliorées.

Le chapitre suivant et dernier chapitre synthétise les résultats du projet et le conclut.

8 Conclusion

Cet ultime chapitre conclut le projet. Il synthétise les résultats puis discute des perspectives du projet. Finalement, il se termine par une conclusion personnelle de l'auteur.

8.1 Résultats et atteinte des objectifs

Le projet a toujours été qualifié de projet de recherche. L'objectif du projet est d'apporter le support d'une infrastructure dynamique pour Kollaps.

Cette fonctionnalité s'est matérialisée sous la forme d'un système d'orchestration permettant de gérer un regroupement de nœuds et de distribuer des tâches en fonction des contraintes de Kollaps dans un environnement de calcul volontaire. Par sa nature, il apporte aussi la possibilité de supporter plusieurs expérimentations simultanées, ce que n'est pas capable de faire l'implémentation actuelle.

Le support d'une infrastructure dynamique pour Kollaps demande de repenser intégralement comment le logiciel fonctionne. Nous passons d'un système prévu pour une seule exécution à un système pouvant fonctionner sur un regroupement de machines qui évolue dans le temps. Ce changement drastique de la conception fondamentale de Kollaps a poussé à une ré-implémentation complète du logiciel.

Aujourd'hui, le nouveau Kollaps est capable d'exécuter les fonctionnalités corps du système original, à savoir l'émulation correcte des expérimentations en supportant la majorité des fonctionnalités plus poussées, comme la gestion des événements. Il offre en plus le support dynamique de l'infrastructure.

Les objectifs que définit le cahier des charges (annexes, ch. 11.1) sont atteints. Cela ne signifie pas que le logiciel est dans sa version définitive. Le chapitre 7 discute des possibilités d'amélioration et les fonctionnalités manquantes pour atteindre le même niveau que le logiciel original.

8.2 Conclusion personnelle

Le projet est passionnant. Il touche à plusieurs domaines des systèmes distribués et, notamment, effleure le répertoire des systèmes de calcul volontaire. J'ai [Romain Agostinelli] beaucoup apprécié travailler sur les différents problèmes qui sont apparus lors du projet. Le domaine des systèmes décentralisés, ainsi que les systèmes concurrents m'intéresse énormément. Le projet m'a permis de constater la difficulté de mise en place d'un système d'orchestration, même avec des capacités réduites. Ces systèmes, par leur nature distribuée, obligent les développeurs à prévoir les pannes et prévoir des procédures de récupération.

Du point de vue de l'implémentation, ce projet est ma première expérience avec le langage Rust. Je maîtrise plusieurs langages, comme Java, C et C++, Scala ainsi que Go. Rust chamboule les codes de la programmation appris depuis des années. Hormis la syntaxe particulière, son Borrow Checker et les règles qu'il impose demande de réapprendre à programmer. Plus que Rust uniquement, Rust avec asynchronicité et coroutines est encore une couche supplémentaire de difficulté. Malgré mon aisance avec l'apprentissage de nouveaux langages, il m'a fallu beaucoup de temps pour me sentir à l'aise. Cependant, je suis très content d'avoir appris ce langage et une fois qu'on arrive à le maîtriser et maîtriser ses codes, il est très satisfaisant à utiliser.

Une autre source de satisfaction, est le fait que dans certains cas, des structures et techniques apprises durant l'année de Master m'ont permis d'apporter de meilleures solutions, par exemple avec l'utilisation de structures sans verrou.

En ce qui concerne le projet lui-même, j'aurais apprécié me concentrer plus sur la mise en place d'un système d'orchestration pour Kollaps, sans devoir réimplémenter son code. Malheureusement, cela aurait sûrement produit uniquement un système d'orchestration qui n'aurait pas été compatible avec la version originale de Kollaps. Il aurait fallu plus de temps afin de rendre ensuite les deux systèmes compatibles et complètement intégrés.

Un dernier point à aborder, est le fait que je suis content du résultat et des efforts qui ont été fournis. Cette nouvelle version de Kollaps permet d'atteindre les objectifs, tout en offrant une base saine pour de futures améliorations. Des solutions à des problèmes importants ont été trouvées, notamment en ce qui concerne les tests de performance et l'orchestration.

Ce fut avec beaucoup de motivation et d'intérêt que j'ai effectué ce projet.

9 Déclaration d'honneur

Je, soussigné, Romain Agostinelli, déclare sur l'honneur que le travail rendu est le fruit d'un travail personnel. Je certifie ne pas avoir eu recours au plagiat ou à toutes autres formes de fraudes. Toutes les sources d'information utilisées et les citations d'auteur ont été clairement mentionnées.

Romain Agostinelli
Villars-sur-Glâne, le 08 février 2023



10 Ressources

Cette section décrit les ressources du projet, les répertoires utilisés ainsi que les dépendances de cette nouvelle version de Kollaps.

Administratif :

Répertoire contenant les documents administratifs concernant la thèse :

<https://github.com/ag0st/kollaps-msc-admin>

Kollaps :

Répertoires de la version actuelle de Kollaps, ainsi que la nouvelle version développée dans ce projet :

- Version actuelle : <https://github.com/miguelammatos/Kollaps>
- Version de ce projet : <https://github.com/ag0st/kollaps>

Expérimentations et applications tierces créées pour ce projet :

Répertoires contenant les différents projets d'expérimentations produits durant la thèse :

- SubDisco : système de découverte de capacité réseau décentralisé. Ensuite intégré à Kollaps sous le nom de CGraph.
Implémentation accessible ici : <https://github.com/ag0st/subdisco>
- NetTest : logiciel de test de performance écrit en Rust. Abandonné au profit d'iPerf3.
Implémentation accessible ici : <https://github.com/ag0st/nettest>

Dépendances du projet :

- Tokio : <https://tokio.rs/>
- Serde : <https://serde.rs/>
- Clap : <https://github.com/clap-rs/clap>
- Uuid : <https://github.com/uuid-rs/uuid>
- RedBPF : <https://github.com/foniod/redbpf>
- Lockfree : <https://gitlab.com/bzim/lockfree/>

Glossary

backbone Le backbone signifie la colonne vertébrale. En télécommunication, ce nom est utilisé pour représenter le corps d'un réseau, sur quoi viennent s'attacher ensuite les périphériques.. 13

JSON JavaScript Object Notation,JavaScript Object Notation (JSON) est un format de données textuelles dérivé de la notation des objets du langage JavaScript. Il permet de représenter de l'information structurée comme le XML par exemple[17].. 35

linux traffic control Linux traffic control est un outil disponible sur Linux permettant de modifier l'utilisation réseau en fonction de certaine description. Par exemple, il permet de modifier la bande passante pour une certaine application ou vers un certain destinataire. Il est l'outil au coeur de Kollaps.. 3, 4, 9, 16, 27, 28, 46–48

PT Processus Terminal. Dans le cadre de ce projet, signifie un processus qui doit être surveillé par une expérimentation Kollaps. Dans la description d'une topologie, il s'agit d'un service.. 27, 28, 30, 46, 47, 49, 52, 53, 55

QoS Quality of Service. Dans un contexte de réseau, il s'agit de la modification forcée des propriétés réseau pour atteindre certains objectifs, souvent la réduction de performance ou la prioritisation de certains paquets.. 3, 9

TCAL Traffic Control Abstraction Layer. Librairie d'abstraction permettant de facilement modifier les valeurs du TC.. 27

thread Un thread est similaire à un processus car tous deux représentent l'exécution d'un ensemble d'instructions du langage machine d'un processeur.[20]. 35, 51, 54, 55

References

- [1] Y. Bejerano et al. "Physical topology discovery for large multisubnet networks". In: *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*. Vol. 1. 2003, 342–352 vol.1. doi: [10.1109/INFCOM.2003.1208686](https://doi.org/10.1109/INFCOM.2003.1208686).
- [2] OpenVSwitch contributors. *Quality of Service*. URL: <https://docs.openvswitch.org/en/latest/faq/qos/>. (accessed: 06.02.2023).
- [3] Docker. *Use IPvlan networks*. URL: <https://docs.docker.com/network/ipvlan/>. (accessed: 01.02.2023).
- [4] eBPF Documentation. URL: <https://ebpf.io/what-is-ebpf>. (accessed: 06.02.2023).
- [5] J. Lau Ed., M. Townsley Ed., and I. Goyret Ed. *Layer Two Tunneling Protocol - Version 3 (L2TPv3)*. RFC 3931. RFC Editor, Mar. 2005. URL: <https://www.rfc-editor.org/rfc/rfc3931.txt>.
- [6] The Apache Software Foundation. *ZooKeeper Recipes and Solutions*. URL: <https://zookeeper.apache.org/doc/current/recipes.html>. (accessed: 06.02.2023).
- [7] Paulo Gouveia et al. "Kollaps: Decentralized and Dynamic Topology Emulation". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. doi: [10.1145/3342195.3387540](https://doi.org/10.1145/3342195.3387540). URL: <https://doi.org/10.1145/3342195.3387540>.
- [8] Riccardo Gusella and Stefano Zatti. "The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3 BSD". In: *IEEE transactions on Software Engineering* 15.7 (1989), pp. 847–853.

-
- [9] Anthony Heddings. *What Are Unix Sockets and How Do They Work?* URL: <https://www.howtogeek.com/devops/what-are-unix-sockets-and-how-do-they-work/>. (accessed: 31.01.2023).
 - [10] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem". In: *Concurrency: the works of leslie lamport*. 2019, pp. 203–226.
 - [11] Luca Liechti et al. "THUNDERSTORM: a tool to evaluate dynamic network topologies on distributed systems". In: *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2019, pp. 241–24109.
 - [12] *namespaces(1) Linux User's Manual*. 5.13. Dec. 2012. URL: <https://man7.org/linux/man-pages/man1/nsenter.1.html>.
 - [13] *namespaces(1) Linux User's Manual*. 5.13. URL: <https://man7.org/linux/man-pages/man7/namespaces.7.html>.
 - [14] J. Postel. *User Datagram Protocol*. RFC 768. RFC Editor, Aug. 1980. URL: <https://www.rfc-editor.org/rfc/rfc768>.
 - [15] DARPA INTERNET PROGRAM. *INTERNET PROTOCOL*. RFC 791. RFC Editor, Sept. 1981. URL: <https://www.rfc-editor.org/rfc/rfc791>.
 - [16] Tigera. *eBPF XDP: The Basics and a Quick Tutorial*. URL: <https://www.tigera.io/learn/guides/ebpf/ebpf-xdp/>. (accessed: 01.02.2023).
 - [17] Wikipédia. *JavaScript Object Notation — Wikipédia, l'encyclopédie libre*. [En ligne; Page disponible le 30-janvier-2023]. 2022. URL: https://fr.wikipedia.org/wiki/JavaScript_Object_Notation.
 - [18] Wikipedia contributors. *Endianness — Wikipedia, The Free Encyclopedia*. [Online; accessed 2-February-2023]. 2023. URL: <https://en.wikipedia.org/wiki/Endianness#References>.
 - [19] Wikipedia contributors. *Leader election — Wikipedia, The Free Encyclopedia*. 2021. URL: https://en.wikipedia.org/wiki/Leader_election. (accessed: 06.02.2023).
 - [20] Wikipedia contributors. *Thread (computing) — Wikipedia, The Free Encyclopedia*. [Online; accessed 3-June-2022]. 2022. URL: [https://en.wikipedia.org/w/index.php?title=Thread_\(computing\)&oldid=1081178886](https://en.wikipedia.org/w/index.php?title=Thread_(computing)&oldid=1081178886).

11 Annexes

11.1 Cahier des charges

Dynamic Infrastructure Support for Kollaps

Thèse de Master - Computer science

Cahier des charges

Fribourg, Septembre 2022

Auteur :

Romain Agostinelli

romain.agostinelli@gmail.com

Mandant :

Université de Neuchâtel

Superviseur :

Pasin Marcelo

marcelo.pasin@hes-so.ch

Personne de contact :

Schiavoni Valerio

valerio.schiavoni@unine.ch

Version: 1.0

Contents

1	Introduction	1
2	Contexte	1
3	Objectif	1
4	Architecture actuelle	1
4.1	Langage de programmation	1
4.2	Composants	1
5	Planning	2
5.1	Première phase	2
5.2	Deuxième phase	3

1 Introduction

Kollaps [2, 3], un projet de recherche conjoint entre l'Université de Neuchâtel et l'IST Lisbonne, est un banc d'essai de recherche expérimentale pour évaluer les systèmes distribués à grande échelle. Kollaps est open-source [1] et a été utilisé avec succès pour l'enseignement des cours de systèmes distribués. En bref, Kollaps permet de déployer des systèmes non modifiés, emballés comme des conteneurs Docker, des machines virtuelles ou des processus normaux, sur des topologies émulées. Les topologies sont ensuite mises en correspondance avec un cluster physique sous-jacent. Un langage de description de domaine permet de décrire les expériences et les topologies testées.

2 Contexte

Les expériences Kollaps se déploient actuellement sur deux orchestrateurs différents (Kubernetes et Docker Swarm) et aussi en mode Baremetal. La topologie de ces orchestrateurs doit être fixe.

Le cluster exécutant la ou les expériences Kollaps est actuellement défini de manière statique et ne permet pas de changements dynamiques.

Toutefois, de tels changements peuvent survenir soudainement, par exemple en raison de défaillances matérielles ou logicielles (entraînant le retrait des nœuds existants), l'expulsion d'instances temporaires (par exemple, des instances ponctuelles de type EC2) ou l'ajout de nouveaux nœuds (offrant des capacités de calcul supplémentaires à utiliser pour améliorer l'équilibre de la charge globale).

Des capacités de calcul supplémentaires peuvent également être fournies par des nœuds proposés dans le cadre d'un modèle de "calcul volontaire" (par exemple, par des clients tiers qui ne rejoignent le réseau Kollaps que pour une durée limitée). Une fois ces changements d'infrastructure effectués, les topologies émulées actuellement en cours d'exécution (et les composants de base sous-jacents de Kollaps) devraient être rééquilibrées.

3 Objectif

Développer et étendre le support d'une infrastructure dynamique pour Kollaps.

4 Architecture actuelle

L'architecture actuelle du projet Kollaps est décrite dans les chapitres ci-dessous. Cette architecture sert de base pour le projet. Elle n'est pas définitive et est sujette à modification si le projet le requiert.

4.1 Langage de programmation

Dans le cadre de la réécriture de l'outil Kollaps en Rust (précédemment écrit en Python), le langage imposé est Rust.

4.2 Composants

Kollaps est actuellement composé de plusieurs composants qui communiquent à travers le réseau et par mémoire partagée pour l'échange d'informations (méta-données) afin de créer une infrastructure décentralisée et cohérente (figure 2).

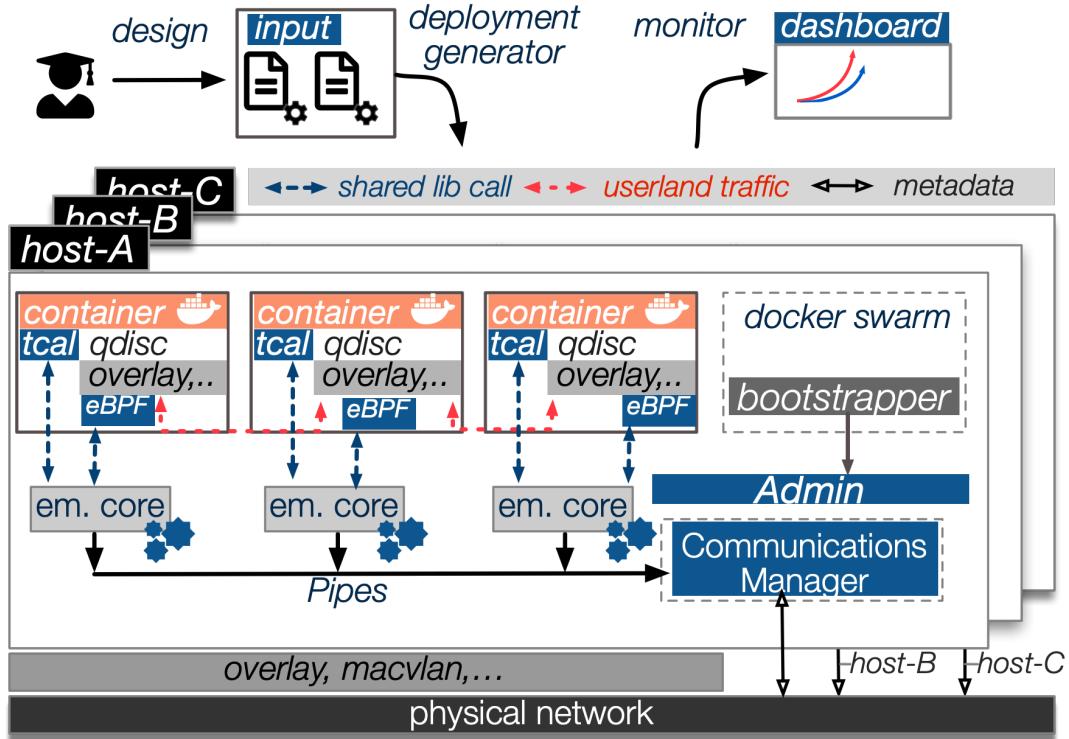


Figure 2: Architecture actuelle de Kollaps

Dans le cadre de ce projet, le but est de garder la même infrastructure générale du programme qui a fait ses preuves.

Il se peut toutefois qu'une partie de l'architecture doivent être modifiée pour atteindre les objectifs du projet de recherche.

Toute modification du code se fera sur un fork du répertoire Git du projet, ou sur un autre projet afin de ne pas endommager le code existant.

5 Planning

Il s'agit d'un projet de recherche et donc, il n'y a pas de planning complètement défini. La direction du projet peut changer en fonction des résultats obtenus et de nouvelles pistes découvertes tout au long de la thèse.

Cependant, pour le commencement, deux premières phases sont définies.

5.1 Première phase

La première phase du projet se concentre sur l'apprentissage. Dans cette phase, l'étudiant doit prendre en main le projet Kollaps, faire des recherches et mettre au point un cahier des charges. Il doit aussi se perfectionner dans le langage Rust afin de pouvoir comprendre et fournir du code à Kollaps.

C'est aussi dans cette première phase qu'il doit mettre en place les outils de communication avec l'équipe et se coordonner avec les autres membres du projet.

Il doit aussi pouvoir mettre en place un environnement de travail afin qu'il puisse ensuite facilement travailler avec Kollaps et tester de nouvelles implémentations.

Tâches :

- Apprentissage de Rust
- Mise en place d'un environnement de développement
- Définition d'un cahier des charges
- Se mettre à niveau des publications scientifiques liées au projet

Délivrible : Environnement de développement + Cahier des charges

Durée : 2 semaines

5.2 Deuxième phase

L'étudiant, après avoir pris en main le projet et fait valider son cahier des charges, doit commencer à faire un travail d'analyse en proposant des solutions au problème.

Il s'agit d'un travail de recherche et d'expérimentation. Il proposera ses idées et ses concepts aux chefs de projet avec qui, il définira ensuite des pistes à creuser.

Tâches :

- Recherche de concepts existant pour la problématique
- Développement de nouveaux concepts
- Expérimentations
- Documentation

Délivrible : Documentation et comparaison de différentes idées permettant de résoudre la problématique

Durée : 2.5 semaines

References

- [1] Kollaps authors. *Kollaps GitHub page*. URL: <https://github.com/miguelammatos/Kollaps>. (accessed: 19.09.2022).
- [2] Paulo Gouveia et al. "Kollaps: decentralized and dynamic topology emulation". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.
- [3] Luca Liechti et al. "THUNDERSTORM: a tool to evaluate dynamic network topologies on distributed systems". In: *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2019, pp. 241–24109.

11.2 Analyse

11.2.1 Ébauche de protocole pour une librairie de test de performance

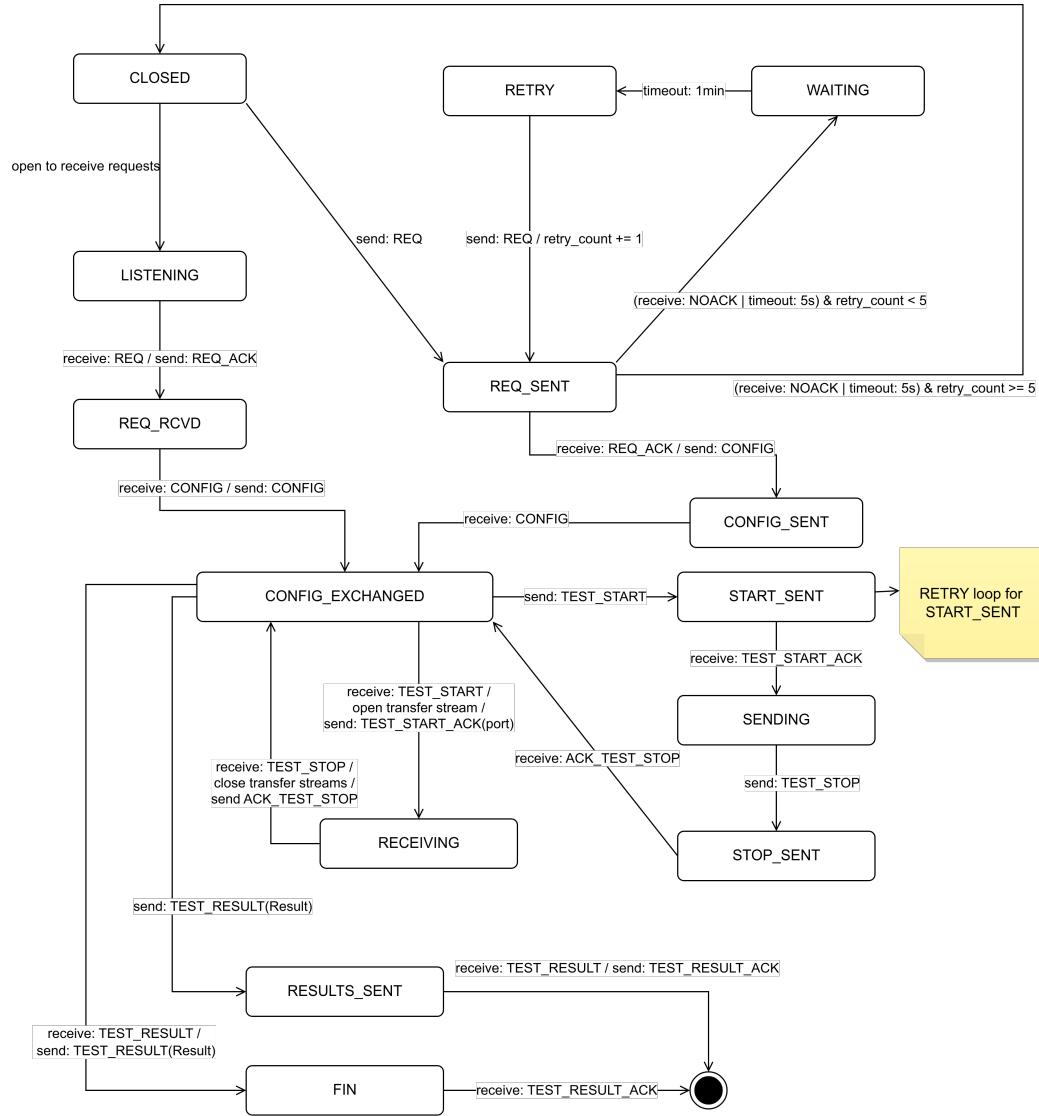


Figure 50: Ébauche d'un protocole de communication pour une librairie de test de performance

11.3 Conception

11.3.1 Agrandissement, système événementiel

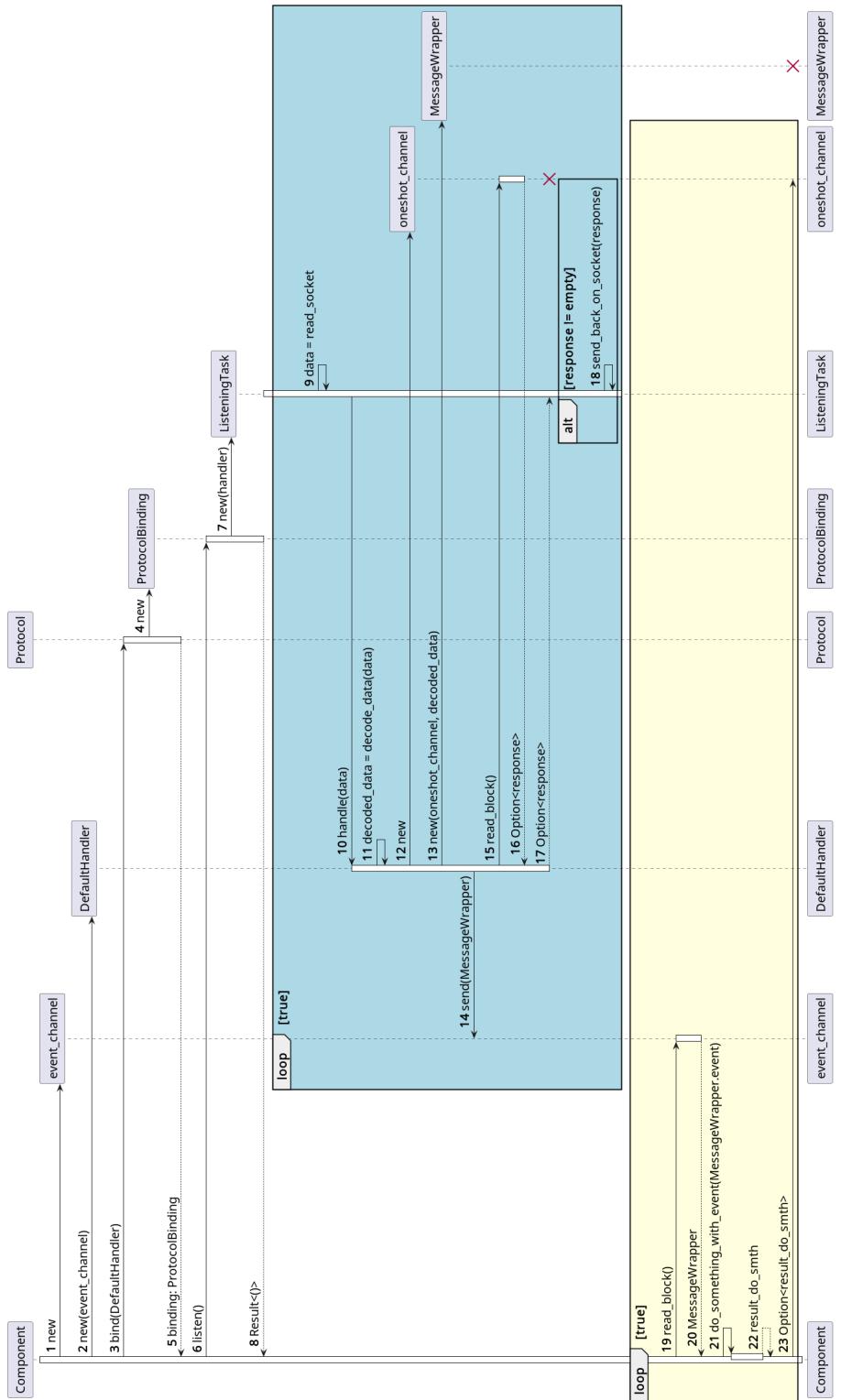


Figure 51: Mise en place d'un système événementiel.

11.3.2 Agrandissement, boucle principale de l'OManager

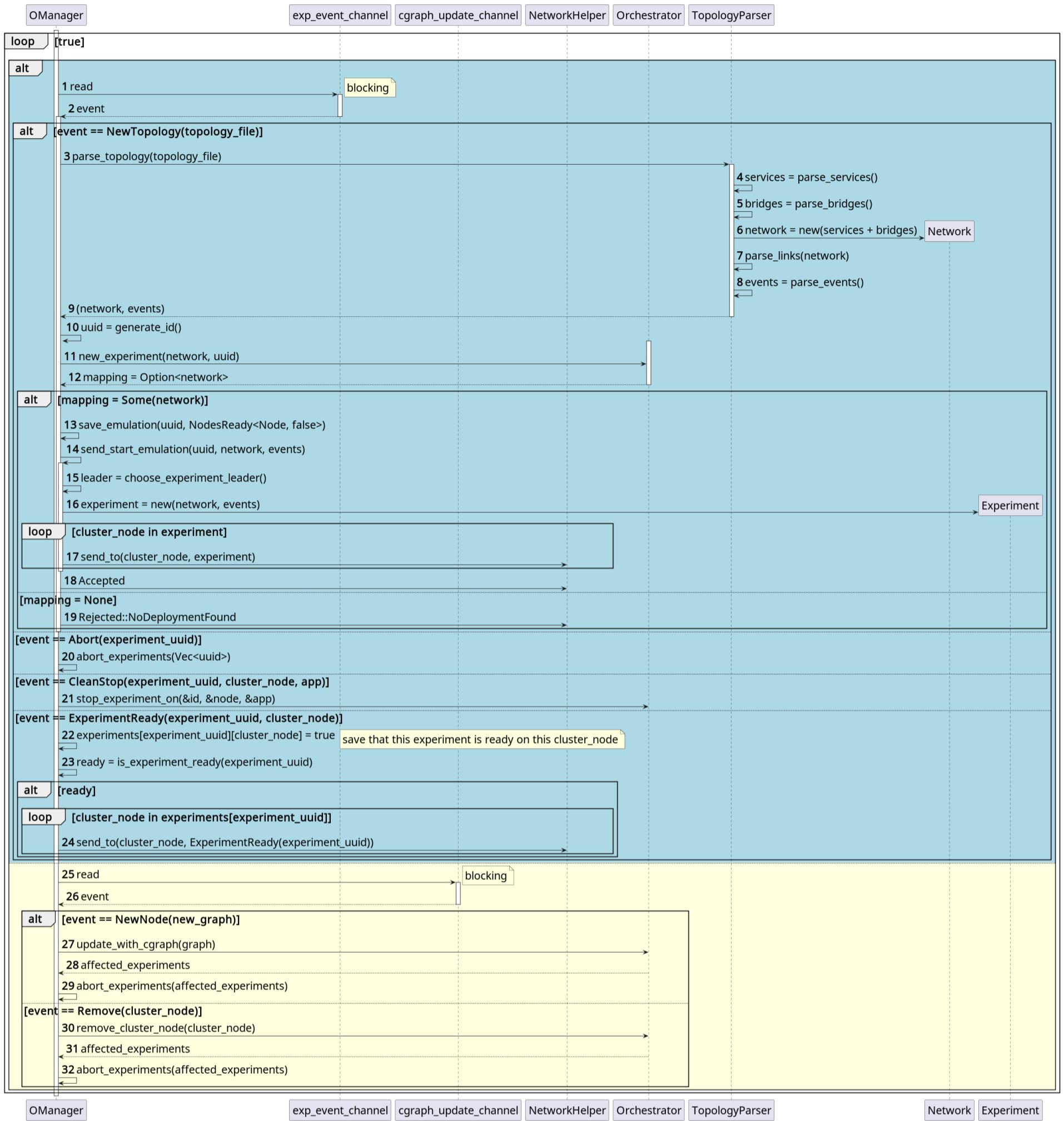


Figure 52: Boucle principale de l'OManager.

11.4 Diagramme de classe des structures

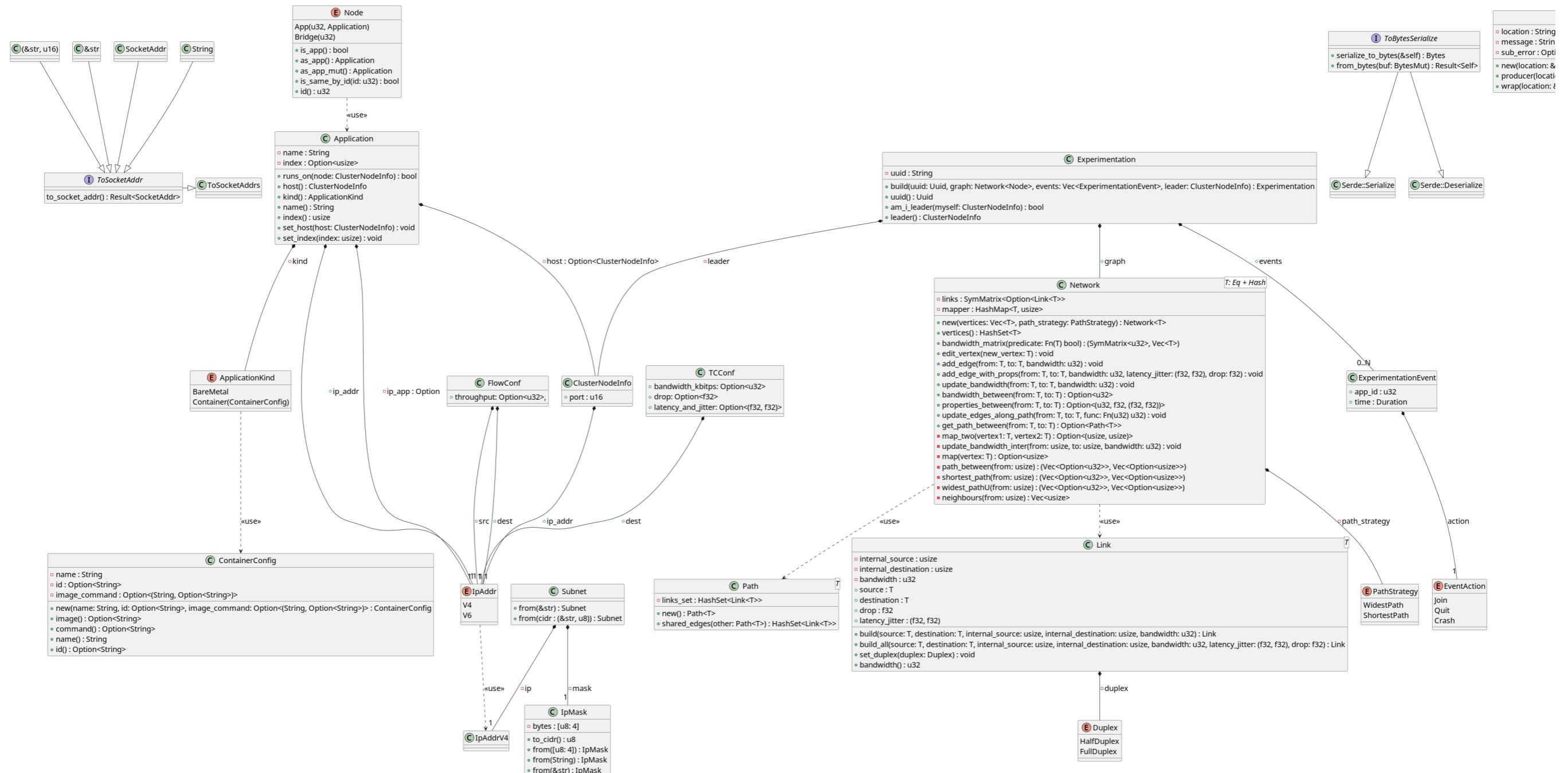


Figure 53: Diagramme de classes des structures indépendantes de Kollaps