Université de Technologie de Compiègne
Printemps 2024

SR05 – Systèmes et Algorithmes Répartis

Gestion d'actions

Étude – Mécanisme de gestion des participants

Auteurs : Hugo Jespierre, Adhavane Moudougannane, Nathan Piteux, Virginie Tio

Responsable de l'UV : Bertrand Ducourthial

18 juin 2024

Ce document synthétise l'ensemble du travail effectué et réflexion menée à la date du 18 juin 2024 sur l'étude conduite dans le cadre de l'UV SR05 : Systèmes et Algorithmes Répartis, à l'Université de Technologie de Compiègne durant le semestre de Printemps 2024.

Introduction

Dans le cadre de notre projet UV SR05, nous avons développé une application simulant la gestion d'un stock d'actions par une banque, liée à des clients souhaitant acheter ou vendre des actions. Un seul type d'action est concerné, et le stock disponible est partagé entre différents sites.

Nous avons mis en place une exclusion mutuelle pour qu'un seul site puisse acheter ou vendre des actions à la fois, évitant ainsi des incohérences comme un stock négatif. Le nombre de clients, fixé au lancement de l'application, n'était pas dynamique. Notre projet s'est concentré sur la gestion des réplicats et des sauvegardes avec un nombre fixe de participants, mais cette approche pose des défis pour intégrer de nouveaux participants ou en retirer sans perturber le système.

La problématique est donc : **comment faire évoluer la liste de participants tout en assurant stabilité et cohérence ?** L'objectif est de mettre en œuvre des mécanismes efficaces de gestion des participants, permettant l'ajout dynamique de nouveaux membres et le retrait sans interruption des existants.

Pour assurer une intégration harmonieuse des nouveaux participants, il est essentiel de considérer des besoins spécifiques, qui incluent les points suivants :

- Développer un mécanisme d'admission fluide pour les nouveaux participants, avec une communication et diffusion des messages d'admission claires.
- Assurer la diffusion de messages avec terminaison explicite pour maintenir la cohérence du système.
- Gérer les conflits, notamment lors d'admissions simultanées, grâce à des mécanismes d'élection et de coordination.
- Gérer le départ des participants sans perturber le réseau, incluant la réallocation des tâches et la reconfiguration.

Cette étude vise à développer une application de gestion des participants, nommée "contrôleur NET", pour une gestion dynamique et robuste des membres du réseau, qui sera ensuite intégrée à notre travail existant.

Nous aborderons donc les sujets suivants dans ce rapport : l'architecture du réseau, les algorithmes de gestion des membres du réseau, l'intégration de notre travail au projet et un mode d'emploi avec un exemple d'utilisation.

I. Architecture du réseau

A. Topologie existante

Le réseau de notre application de gestion de stock était structuré en un anneau unidirectionnel, avec une application de contrôle correspondant à la structure numéro 7 de l'Étape 3 du projet. Les communications suivaient le principe FIFO (*First In, First Out*). La figure 1 illustre le réseau réalisé dans le cadre du projet précédent :

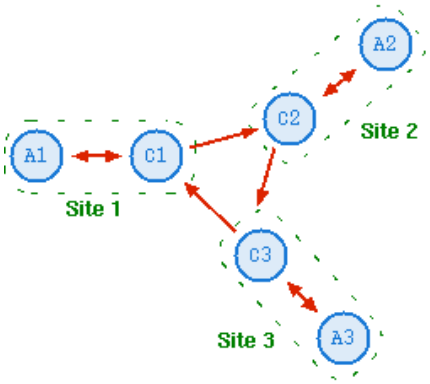

Figure 1. Structure numéro 7 de l'Étape 3 du projet

Une application de contrôle dans un système distribué supervise et coordonne les activités des différents composants. La différence entre deux programmes distincts réside dans leurs rôles : l'application gère les fonctionnalités spécifiques du site, tandis que le contrôleur supervise l'ensemble du système. Séparer ces fonctionnalités permet un découplage clair et une meilleure gestion des dépendances, favorisant ainsi la modularité et la maintenance du système.

Nous avons mis en place cette architecture en utilisant deux programmes distincts en Go : l'application (`app`) et le contrôleur (`ctl`), correspondant respectivement à l'application et au contrôleur dans le système distribué. Les communications entre ces deux composants étaient gérées en shell en redirigeant les entrées/sorties standard vers des pipes interconnectées.

L'utilisation d'un anneau est une approche efficace pour implémenter les algorithmes de calcul d'horloge logique, de file d'attente répartie et de construction d'instantanés dans un système distribué. Cependant, la topologie en anneau peut être considérée comme simpliste, présentant des limitations en termes de redondance et de tolérance aux pannes, ainsi que des défis pour l'équilibrage de charge et la scalabilité dans les environnements distribués à grande échelle.

L'enjeu de l'étude réside dans la gestion dynamique de l'ajout et de la suppression de participants, ce qui signifie pouvoir insérer ou retirer des nœuds de l'anneau pendant l'exécution des programmes. Dans ce contexte, la topologie en anneau pose plusieurs défis à résoudre. **Comment pouvons-nous modifier notre réseau pour le rendre dynamique, permettant ainsi l'insertion et la suppression de nœuds durant l'exécution des programmes ?**

B. Solutions de réseau dynamique

Nous disposons de plusieurs solutions pour rendre notre réseau dynamique, permettant ainsi l'insertion et la suppression de nœuds en cours d'exécution.

1. Modification physique de l'anneau unidirectionnelle

Une première approche consiste à modifier physiquement l'anneau en conservant sa topologie actuelle. Lors de l'ajout ou de la suppression d'un nœud, on pourrait "simplement" casser et/ou recréer des *pipes* pour insérer le nouveau nœud dans l'anneau. Par exemple, lors de l'ajout d'un nouveau nœud, on pourrait casser le *pipe* reliant les nœuds voisins, puis créer de nouveaux *pipes* pour inclure le nouveau nœud dans l'anneau. Cette approche nécessite une gestion minutieuse des connexions et peut entraîner des interruptions de service pendant le processus de modification de l'anneau.

Cette solution nous aurait permis de conserver notre structure actuelle et de garantir que les algorithmes fonctionnent comme d'habitude. Elle représente l'approche la plus simple et directe pour gérer dynamiquement l'ajout et la suppression de nœuds dans l'anneau.

2. Réseau logique au-dessus du réseau physique

La première solution, bien que simple conceptuellement, peut ne pas être optimale en pratique. En effet, à chaque ajout ou suppression de nœud, elle implique des manipulations au niveau physique, telles que des redirections d'entrées/sorties, qui peuvent être coûteuses en termes de performance.

Une idée intéressante serait de mettre en place un réseau logique au-dessus du réseau physique pour gérer les communications. Dans ce scénario, chaque nœud aurait dorénavant connaissance de ses voisins grâce à ce réseau logique. Cette approche permettrait de déléguer la gestion des connexions à un niveau supérieur, offrant ainsi une abstraction plus élevée et une flexibilité accrue pour gérer dynamiquement l'ajout et la suppression de nœuds. Chaque nœud pourrait alors maintenir une liste de ses voisins logiques, facilitant ainsi les opérations de communication et d'administration du réseau.

Imaginons un réseau de quatre nœuds disposés en anneau physique, numérotés de 0 à 3 dans le sens horaire. Chaque nœud est connecté à ses voisins physiques par des pipes pour les communications. Maintenant, introduisons un réseau logique au-dessus du réseau physique. Chaque nœud, en plus de ses connexions physiques, dispose d'une connaissance de ses voisins logiques. Initialement, le réseau physique est identique au réseau logique. Par exemple, le nœud 0 a pour voisins logiques les nœuds 3 et 1.

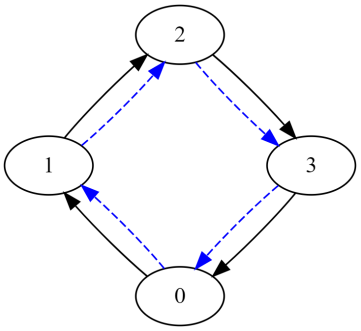

Figure 2. Anneau unidirectionnel de quatre nœuds
(en noir, les connexions physiques, et en bleu, les connexions logiques)

Lorsqu'un nouveau nœud est ajouté, il peut se connecter aux nœuds voisins en utilisant les informations du réseau logique, sans avoir à modifier les connexions physiques entre les autres nœuds. Par exemple, si nous ajoutons un nœud 4 au nœud 0, nous le définissons simplement comme un nouveau voisin logique du nœud 0, et vice versa. Ensuite, nous devons mettre à jour les voisins logiques du nœud 0 en remplaçant l'un des anciens voisins par le nœud 5. De manière symétrique, nous mettons à jour les voisins logiques du nœud voisin ajouté en remplaçant le nœud 0 par le nœud 5.

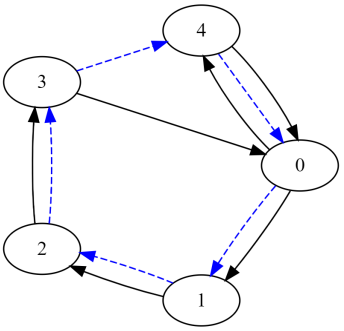

Figure 2. Ajout du nœud 4 dans l'anneau unidirectionnel
(en noir, les connexions physiques, et en bleu, les connexions logiques)

Cette approche est assez performante car elle permettrait de conserver un réseau logique en anneau dans lequel nos algorithmes pourraient fonctionner. Cependant, un inconvénient majeur est la visibilité accrue des nœuds sur leurs voisins. Avant, chaque nœud était indépendant de ses voisins parce qu'il ne connaissait pas la direction de sa sortie ni qui lui transmettait un message.

3. Changer la topologie

Les deux solutions sont réalisables, mais l'anneau semble trop simpliste et peu performant pour ce problème, notamment en termes de nombre de messages. La dernière idée, bien que plus complexe, consistait à revoir complètement notre structure de réseau en utilisant un arbre bidirectionnel. Bien que cette approche soit plus difficile à mettre en œuvre, elle est plus réaliste et permet d'envoyer moins de messages.

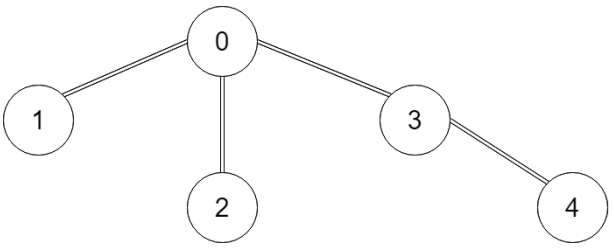

Figure 3. Arbre bidirectionnel

C. Arbre bidirectionnel

Dans un anneau unidirectionnel, les messages ne transitaient jamais indéfiniment de site en site. On était assuré que le message atteindrait toujours le site qui est dans l'anneau. Ainsi, lorsqu'un site recevait un message qui ne lui était pas destiné, il suffisait de le remettre dans l'anneau. Avec un arbre bidirectionnel, c'est plus complexe car si nous retransmettons un message qui ne nous est pas destiné, il faut éviter de le recevoir à nouveau, sinon nous aurions des messages qui circuleraient indéfiniment entre deux sites.

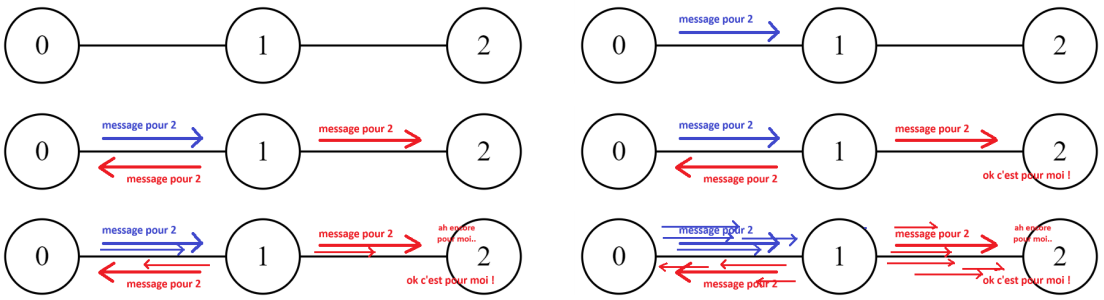

Figure 4. Messages qui bouclent à l'infini sans un réseau bidirectionnel

Lorsqu'un nœud reçoit un message dans le réseau de l'application et qu'il n'est pas le destinataire prévu, il vérifie s'il est le parent du message, c'est-à-dire s'il était le précédent transmetteur du message. Dans ce cas, cela signifie qu'il avait déjà reçu le message et qu'il l'avait transmis à son voisin direct. Par conséquent, il peut en toute sécurité ignorer le message pour éviter les boucles infinies de transmission.

Si le nœud n'est pas le parent du message, cela signifie que c'est la première fois qu'il reçoit le message. Il doit alors vérifier si le message lui est destiné ; s'il l'est, il le traite. Sinon, il retransmet le message en spécifiant le parent du message, qui est le transmetteur qui lui a envoyé le message, et il complète également le champ `transmitter` avec son propre site. Ainsi, le transmetteur précédent, grâce à notre topologie d'arbre bidirectionnel, saura qu'il est le parent et pourra ignorer le message. De la même manière, les autres nœuds auront connaissance du transmetteur actuel pour éventuellement procéder de la même manière. Ce processus se répète jusqu'à ce que le message atteigne le bon destinataire.

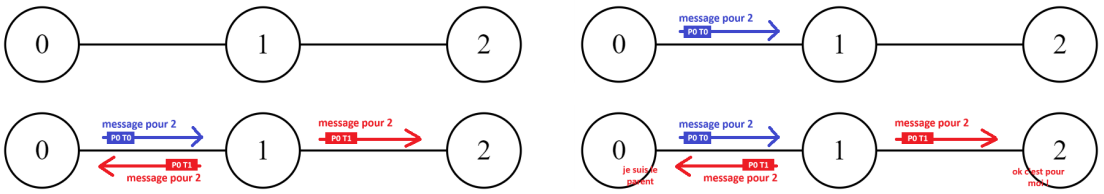

Figure 5. Système du parent/transmetteur pour ignorer les messages rebonds

Ce système de parent et de transmetteur assure la terminaison des messages et évite les boucles infinies.

Bien que l'idéal serait de renvoyer le message à tous les voisins sauf au transmetteur précédent, les connexions bidirectionnelles impliquent que le transmetteur précédent recevra le message de toute façon. Par conséquent, nous devons fournir les informations nécessaires au transmetteur précédent pour qu'il sache qu'il était le transmetteur initial du message et qu'il n'a pas besoin de le retraiter. C'est là que le concept de parent entre en jeu.

Dans le réseau en arbre, l'absence de cycles est essentielle pour assurer un fonctionnement correct du système. Un arbre est par définition un graphe acyclique et connexe.

II. Algorithmes de gestion de membres du réseau

A. Ajout de site

Le fichier `addsite.sh` du dossier `scripts` permet l'ajout d'un nouveau site dans le réseau. Ce fichier est exécuté directement dans un terminal lorsque l'ajout d'un site est voulu. Son fonctionnement est détaillé ci-dessous :

- **Initialisation :** Le script est lancé avec deux arguments : le site parent et l'identifiant du nouveau site à ajouter. Par exemple, la commande `./scripts/v3/addsite 3 5` ajoute le site 5 sous le site 3.
- **Création des connexions :** Au lancement du script, des pipes de connexion sont créées, ainsi que la *map* (fichier dans le dossier `/tmp`) contenant la liste des sorties du nouveau site. Le nouveau site est alors connecté à son contrôleur (`ctl`) et à son application de manière bidirectionnelle. Ensuite, avec le script, la connexion du `NET` du nouveau site à celui du site parent est établie de manière unidirectionnelle.
- **Élection :** Une fois les connexions initiales créées, une élection est lancée pour valider l'ajout du nouveau site. Cette étape sera détaillée plus bas dans cette partie.
- **Validation et connexion finale :** Après la terminaison et l'approbation de l'élection, le script `addsite_accepted.sh` est exécuté pour établir la connexion définitive entre le site parent et le nouveau site. À l'origine, les sorties du site parent ont été définies avec une commande `tee` pour pouvoir rediriger les *outputs* sur plusieurs

sorties. Cependant, une commande `tee` ne peut pas être exécutée pour redéfinir les sorties par-dessus une autre. Les processus `cat` et `tee` existants sont donc tués. Ensuite, une nouvelle commande `tee` est exécutée, comprenant les anciennes sorties du parent, et ajoute la nouvelle connexion vers le nouveau site pour établir un lien bidirectionnel entre ces deux sites.

B. Suppression de site

Le fichier `removesite.sh` gère la suppression d'un site et prévoit deux scénarios :

1. **Site feuille :** Si le site est une feuille, c'est-à-dire qu'il est connecté à un seul autre site, il est entièrement supprimé du réseau. Une reconfiguration des connexions est donc effectuée pour maintenir l'intégrité du réseau. Cela inclut la suppression des pipes de connexion et du fichier *map*.

2. **Site intermédiaire :** Si le site est au milieu de l'architecture (autrement dit, s'il est connecté à plusieurs sites), il est mis en sommeil. L'attribut `Active` du site est alors mis à `False`, ce qui le rend passif. Cela veut dire que lorsque le site reçoit un message d'un autre site du réseau, il transmet simplement les messages aux autres sites sans les traiter.

C. Élection

Pour l'élection, l'algorithme d'élection par extinction de vagues (Algorithme 22 du cours 9 de SR05) a été repris. Cet algorithme permet d'élire le site candidat qui a le plus petit identifiant. Il est implémenté en utilisant plusieurs étapes clés, chacune reposant sur des messages spécifiques échangés entre les sites du réseau. Voici ci-dessous une description de son implémentation.

1. Initialisation des sites

Chaque `NET` d'un site du réseau est initialisé avec des attributs nécessaires tels que son identifiant (`Id`), son parent (`Parent`), son nombre de voisins (`NbVoisinsAttendus`) et son élu (`Elu`). Au début, les attributs `Parent` et `Elu` sont initialisés à `-1` et à `+∞` respectivement. Pour information, dans le cadre de l'élection, `Parent` sera mis à jour une seule fois lorsqu'un site reçoit un premier message `ELECTION_BLEU`. Cela va référencer son parent à qui il pourra répondre lorsqu'il aura reçu les informations de tous ses fils. Cette valeur ne sera modifiée que s'il reçoit un message provenant d'une autre élection qui serait privilégiée.

2. Déclenchement de l'élection

L'élection peut être déclenchée par deux types de demandes :

- **Demande d'admission** (`DEMANDE_ADMISSION`) : lorsqu'un nouveau site veut rejoindre le réseau.

- **Demande de départ** (`DEMANDE_DEPART`) : lorsqu'un site existant veut quitter le réseau.

3. Propagation de la vague (message `ELECTION_BLEU`)

Le site recevant la demande d'admission ou de départ devient alors un site candidat et commence une élection. Il envoie alors un message `ELECTION_BLEU` à tous ses voisins. Ce message contient l'identifiant du candidat (donc l'identifiant du site courant) qui demande d'être élu.

Chaque site réceptionnant ce message compare l'identifiant du candidat avec son propre identifiant d'élu courant :

- Si le candidat du message a un identifiant plus petit, le site met à jour son élu et continue de propager le message `ELECTION_BLEU` à ses voisins.
- Sinon, le site ignore le message.

4. Remontée de la vague (message `ELECTION_ROUGE`)

Lorsque la propagation de la vague atteint des sites qui n'attendent plus de message de leurs voisins (c'est-à-dire que l'attribut `NbVoisinsAttendus` est égal à 0), ces sites envoient un message `ELECTION_ROUGE` à leur parent.

Les messages `ELECTION_ROUGE` remontent vers l'initiateur de l'élection, confirmant ainsi la couverture complète de la vague.

5. Finalisation de l'élection

L'initiateur de l'élection attend de recevoir tous les messages `ELECTION_ROUGE` de ses voisins. Une fois tous les messages reçus, l'initiateur devient officiellement l'élu. Ce site a donc la permission d'accepter un nouveau site ou de quitter le réseau.

De plus, en fonction du type d'élection (ajout ou suppression de site), des actions spécifiques sont exécutées :

- **Ajout de site :** Après l'approbation de l'élection, un message de type `ACCEPTATION_AJOUT` est envoyé au nouveau site pour le prévenir qu'il a le droit d'être ajouté au réseau. À la réception de ce message, le script `addsite_accepted.sh` est appelé pour compléter le réseau et le nouveau site transmet un message de type `AJOUT` à tous les sites du réseau pour les prévenir qu'il appartient désormais au réseau. L'horloge vectorielle est aussi mise à jour pour inclure le nouveau site.
- **Suppression de site :** Lors de la suppression d'un site, un message de type `SUPPRESSION` est envoyé à tous les autres sites par le site à supprimer pour les prévenir de son départ. En même temps, un message de type `PREVENIR_VOISINS` est envoyé uniquement aux voisins du site à supprimer pour qu'ils puissent

décrémenter leur nombre de voisins en local et ajuster le nombre de messages attendus dans le cas d'une nouvelle élection. L'horloge vectorielle est aussi mise à jour pour refléter ce changement.

Les messages `AJOUT` et `SUPPRESSION` sont diffusés dans tout le réseau pour prévenir les autres NET, qui vont à leur tour prévenir leur contrôleur. À la réception de ce message, l'information qu'un site a réussi une élection est obtenue et les valeurs des variables liées à l'élection sont donc réinitialisées pour pouvoir en démarrer une nouvelle si besoin.

L'algorithme d'élection assure ainsi la coordination lors de l'ajout ou de la suppression de sites. Il permet de gérer les demandes uniques d'ajout et de départ de site, mais surtout les demandes simultanées. Dans le cas d'une demande simultanée, le site avec l'identifiant le plus petit remportera les élections et sera l'élu.

D. NET (*Network Controller*)

Le fichier `net.go` implémente le contrôleur `NET`, qui est essentiel pour gérer dynamiquement et de manière robuste le réseau des participants. Ce contrôleur permet l'ajout et la suppression de sites, en utilisant des messages pour la communication entre les différents nœuds du réseau. L'algorithme d'élection par extinction de vagues, inclus dans ce fichier, est spécialement adapté pour ces opérations dans un réseau distribué.

Parmi les variables globales définies, `NB_SITES` spécifie le nombre de sites dans le réseau, tandis que `liste_sites` contient les identifiants des sites actifs. La variable `p_nb_neighbours` est initialisée plus tard pour indiquer le nombre de voisins pour chaque routeur.

La structure principale, `Net`, représente un site avec plusieurs attributs clés, tels que `Id` pour l'identifiant du routeur, `Active` pour indiquer son état, `Vi` pour le vecteur d'horloge, et d'autres comme `SitetoAdd`, `Parent`, `TypeElection`, `NbVoisinsAttendus`, et `Elu`, qui sont utilisés pour gérer les processus d'élection et de communication entre les routeurs.

Le fichier inclut plusieurs fonctions utilitaires importantes :

- `removeVectorClock(slice []int, s int)`: Retire un élément du vecteur d'horloge.
- `removeSite(slice []int, n int)`: Retire un site de la liste des sites.
- `getFileLineNumber(filename string)`: Compte le nombre de lignes dans un fichier pour initialiser le nombre de voisins. Cette information est essentielle pour l'algorithme d'élection.

L'algorithme d'élection, géré par la fonction `election`, est central pour l'ajout ou la suppression de sites. Le `NET` traite différents types de messages relatifs à ce processus, notamment `AJOUT`, `SUPPRESSION`, `ACCEPTATION_AJOUT`, et `PREVENTION_VOISINS`, qui permettent de coordonner les actions des différents nœuds du réseau de manière cohérente et ordonnée.

III. Intégration au projet

Après avoir développé le fichier `NET` pour gérer les notions d'ajout et suppression de site dans le réseau de façon indépendante, il fallait gérer la connexion entre chaque `NET` et son contrôleur et application.

Dans un premier temps, les scripts ont été modifiés pour prendre en compte la topologie. Une nouvelle connexion bidirectionnelle a été créée pour chaque site et la sortie vers le contrôleur a donc été rajoutée au fichier MAP, pour ne pas oublier de reconnecter un `NET` à son contrôleur en cas de changement dans le réseau.

Ensuite, les fichiers `addsite.sh`, `addsite_accepted.sh` et `removesite.sh` ont été modifiés en conséquence pour afficher ou supprimer les terminaux des contrôleurs et des applications.

Enfin, le plus gros travail a été de modifier le fichier `ctl.go` qui comprend le contrôleur. Toutes les structures statiques, comme les tableaux, sont devenus dynamiques pour accueillir de nouveaux sites si besoin. La liste des sites présents dans le réseau est transmise par le `NET` et stockée localement sur chaque site du réseau. Ainsi, on peut plus facilement itérer sur un tableau et ne considérer que les cases des sites qui sont encore dans le réseau par exemple.

Un gros problème se posait aussi pour les horloges vectorielles, comment faire varier le vecteur sans pour autant se retrouver avec des valeurs incohérentes ? Il a été décidé de laisser l'estampille d'un site dans le vecteur même s'il quittait le réseau pour éviter un décalage des valeurs et une incohérence des indices. Cette valeur ne va plus évoluer, et cela nous permettra éventuellement de savoir combien de messages le site a traité avant de quitter le réseau.

Ainsi, les messages envoyés d'un contrôleur à un autre passent par le NET, mais ne sont pas considérés par ce dernier, vu que le type contenu dans le message ne correspond pas ; il sert juste de relais.

IV. Mode d'emploi et exemples d'utilisation

A. Utilisation

Pour lancer l'application, ouvrez un terminal à la racine du dossier : `cd sr05_actions`. Avant de commencer, assurez-vous que les scripts du répertoir `/scripts` ont les droits d'exécution. Si ce n'est pas le cas, utilisez la commande suivante : `chmod +x scripts/*`. De plus, l'application nécessite le paquet `xterm`, disponible dans les distributions Linux (installable avec la commande `sudo apt install xterm`), ainsi que l'installation de Go. Le programme a été testé avec Go versions 1.22 et 1.23.

Une fois cela fait, exécutez la commande suivante : `./scripts/v3/runner.sh`. Ce script lance l'application de gestion d'actions sur plusieurs sites différents. Comme évoqué précédemment, la structure d'un site est composée d'une application qui interagit avec un contrôleur, lequel interagit avec un net. Les nets des différents sites sont connectés pour former un réseau en arbre binaire tel que défini ci-dessous.

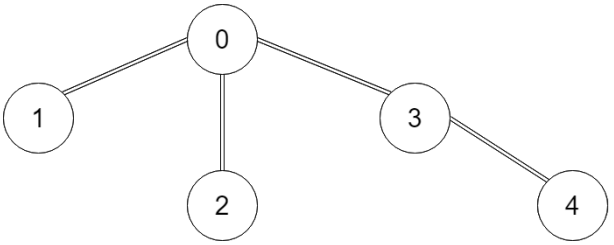

Figure 6. Réseau initial lancé depuis le script `./scripts/v3/runner.sh`

Le lancement du script ouvre donc trois terminaux par site pour la simulation. Chaque terminal possède un titre : `Application x`, `Controleur x` ou `Net x`, où `x` représente l'ID d'un site. Une application, un contrôleur et un net avec le même ID sont liés entre eux.

L'utilisateur va être amené à interagir avec les terminaux se nommant "Application". Les contrôleurs ne sont là que dans un but informatif pour donner un aperçu sur les messages traversant le système. De même, les `NET` permettent de retracer les exécutions propres aux communications inter-site, notamment dans le cadre de l'ajout et la suppression de participants.

B. Fonctionnalités principales

Il retrouvera dans les terminaux d'applications un menu à choix, identique à celui réalisé dans le projet. Le terminal admin (avec l'id 0) possède des fonctionnalités supplémentaires qui permettent de réaliser des calculs d'instantanés.

Pour fermer l'application et tous ses terminaux, l'utilisateur peut se rendre dans le terminal depuis lequel il a lancé la première commande et appuyer sur `Ctrl + C`. Cela permettra d'arrêter tous les processus lancés par le script.

Pour ajouter un site avec l'identifiant `id_noeud` au nœud parent `noeud_parent`, placez-vous à la racine du projet et exécutez la commande suivante : `./scripts/v3/addsite.sh <noeud_parent> <id_noeud>`. Par exemple, pour ajouter un site avec l'identifiant 5 au nœud parent 0, vous pouvez exécuter : `./scripts/v3/addsite.sh 0 5`. Cela ajoutera un nouveau site avec l'identifiant 5, connecté au nœud parent 0 dans votre réseau d'arbres binaires.

Il est possible de tester l'ajout simultané de deux participants et de constater que seul l'un des deux est accepté dans le réseau grâce à l'algorithme d'élection. Simuler l'ajout simultané de deux sites n'est pas facile en raison des différences de vitesse de démarrage

des applications. Cependant, dans la plupart des cas, l'utilisation de l'opérateur `&&` fonctionne pour ajouter deux sites en même temps. Par exemple, en utilisant la commande suivante : `./scripts/v3/addsite.sh 1 6 & ./scripts/v3/addsite.sh 0 7`, cela ajoutera le nœud 6 au site 1 et le nœud 7 au site 0. Dans ce cas, les sites 0 et 1 vont tous les deux faire une demande d'élection et étant donné que 0 est le plus petit, le nœud 7 sera admis dans le site.

Il est difficile de simuler la simultanéité parfaite lors de l'ajout de sites car les applications ne se lancent pas toutes à la même vitesse. Dans la plupart des cas, l'utilisation de l'opérateur `&&` fonctionne bien : le site avec le plus petit identifiant remporte généralement l'élection et est accepté dans le réseau. Cependant, dans environ 1 sur 10 cas, une des commandes est lancée avant l'autre, empêchant ainsi l'un des sites de se porter candidat à temps. Cela n'est pas un problème logique de l'application répartie, mais plutôt une difficulté liée à la simulation d'ajouts ou de suppressions simultanés.

Pour la suppression d'un site, il suffit de sélectionner l'option correspondante dans le menu en appuyant sur 5. Cela entraîne la disparition du terminal, du contrôleur et du réseau associés également.

Lorsqu'un nœud non-feuille quitte, par exemple le nœud 0, le net continue de fonctionner avec le nœud devenant un "zombie". Cela signifie que le réseau reste actif avec ce nœud agissant toujours comme routeur, tandis que les contrôleur et application associés disparaissent.

La suppression simultanée fonctionne de la même manière que l'ajout simultané grâce à l'algorithme d'élection.

Conclusion

Le but de cette étude était d'adapter notre projet pour pouvoir gérer une évolution du nombre de participants dans le réseau, en mettant en place un contrôleur NET pour l'admission fluide, la gestion des conflits via un algorithme d'élection, et la reconfiguration du réseau lors du départ de participants.

Nous avons présenté dans ce rapport les aspects clés de cette évolution, incluant l'architecture du réseau, les algorithmes de gestion des membres, l'intégration de notre travail au projet, et un mode d'emploi avec un exemple d'utilisation.

Grâce à ces nouvelles fonctionnalités, notre application peut désormais s'adapter aux changements de participants sans interruption de service, offrant ainsi une solution plus efficace pour la gestion de stocks d'actions.