IUT Nancy Charlemagne

Université de Lorraine

Département Informatique

Implémentation du protocole de membership SWIM à un logiciel d'édition collaborative de texte

Rapport de stage de DUT informatique

Mendez-Porcel Tom

Année universitaire 2019-2020

Loria

Laboratoire lorrain de recherche en informatique et ses applications

IUT Nancy Charlemagne
Université de Lorraine
2 ter boulevard Charlemagne
BP 55227 54052
Nancy Cedex

Département informatique

Implémentation du protocole de membership SWIM à un logiciel d'édition collaborative de texte

Rapport de stage de DUT informatique
Entreprise : LORIA
Equipe COAST

Mendez-Porcel Tom
Tuteur : Matthieu Nicolas
Parrain de stage : Serguei Lenglet
Année universitaire 2019-2020

Remerciements

J’aimerais remercier mon tuteur, Matthieu Nicolas, et mon co-tuteur, Victorien Elvinger, pour leur encadrement lors de mon stage. En cette période très perturbée, ils n’ont cessé de s’assurer que j’avais aussi bien le matériel que les supports théoriques pour comprendre les concepts et réaliser mon travail. Leurs explications, conseils et retours fréquents ont été essentiels pour le mener à terme.

J’aimerais aussi remercier toute l’équipe COAST pour leur accueil bien que je n’aie peut-être pas pu aussi bien les connaître que je l’aurais souhaité si j’avais pu me rendre physiquement au laboratoire.

Enfin, je voudrais exprimer toute ma reconnaissance envers les enseignants, la direction et le secrétariat de l’IUT pour leur réactivité en ce qui concerne les stages pendant le confinement et tout particulièrement Serguei Lenglet qui a suivi le mien.

Sommaire

Introduction

L'équipe COAST développe depuis plusieurs années MUTE, un éditeur de texte collaboratif temps réel pair-à-pair, qui sert de plateforme de démonstration de ses travaux de recherche.

Dans les applications collaboratives, les utilisateurs ont besoin de plusieurs informations afin de collaborer de façon efficace. Une de ces informations est tout simplement la liste des collaborateurs actuels.

MUTE propose déjà cette fonctionnalité, mais l'implémentation actuelle souffre de plusieurs défauts. Il est donc nécessaire d'en changer.

La solution choisie a été de remplacer l'implémentation existante par une implémentation du protocole de membership SWIM. Mon rôle a été de réaliser une implémentation de ce protocole et de l'intégrer au sein de MUTE.

Ce rapport est organisé de la manière suivante : dans une 1ère partie, nous présentons plus en détails le contexte du stage. Puis nous présentons les concepts nécessaires à la compréhension du sujet dans une 2nde partie. Dans une 3ème partie, nous présentons le prototype d'application intégrant le protocole SWIM que nous avons implémenté. Dans une 4ème partie, nous présentons l'intégration de notre implémentation de SWIM dans MUTE. Puis finalement nous présentons divers axes d'améliorations pour notre implémentation.

Présentation de l'entreprise

Présentation du LORIA

Le LORIA (Laboratoire Lorrain de Recherche en Informatique et ses Applications) est une Unité Mixte de Recherche composée du Centre National de la Recherche Scientifique (CNRS), de l'Université de Lorraine, et de l'Inria. Créé en 1997 le LORIA s'engage dans la recherche fondamentale et appliquée en sciences informatiques. Le centre de Nancy est composé de 28 équipes de recherches réparties dans les 5 départements suivants :

— Département 1 : Algorithmique, calcul, image et géométrie

— Département 2 : Méthodes formelles

— Département 3 : Réseaux, systèmes et services

— Département 4 : Traitement automatique des langues et des connaissances

— Département 5 : Systèmes complexes, intelligence artificielle et robotique

Source : Site internet du LORIA [1]

Présentation de l'équipe COAST

L'équipe COAST s'intéresse au développement de services pour l'hébergement d'équipes et d'entreprises distribuées (ou virtuelles) sur Internet. Les services considérés incluent des services de partage d'objets, de communication, de gestion de tâches, de maintien d'une conscience de groupe, d'aide à la prise de décisions.

L'équipe s'intéresse plus particulièrement aux applications de co-conception et/ou de co-ingénierie pour des domaines variés (Génie Logiciel, Architecture, Formation-Apprentissage).

Ses axes thématiques sont les suivants :

- Systèmes collaboratifs distribués
- Gestion des processus "business" et service informatique
- Interopérabilité et modélisation d'entreprise

L'équipe COAST est composée d'une chargée de recherche, de huit membres de facultés (professeurs ou maîtres de conférences), de 8 étudiants en thèse, d'un ATER et de trois assistantes administratives.

Cadre de travail

En raison des circonstances sanitaires lors de la réalisation du stage, celui-ci s'est déroulé uniquement en télétravail. J'ai travaillé depuis mon ordinateur personnel sur une session Ubuntu comme demandé par mes tuteurs. Pour le travail à distance, un serveur discord à été créé spécifiquement pour l'équipe COAST. J'ai eu des réunions informelles avec mes tuteurs sur celui-ci de manière quasi-hebdomadaire. J'ai également participé aux réunions d'équipe hebdomadaires chaque vendredi. J'ai donc eu la chance de pouvoir travailler dans des conditions favorables tant au niveau du matériel que de l'atmosphère de travail même si je n'ai pas pu me rendre au laboratoire.

Pour la réalisation du code, j'ai utilisé le logiciel « visual studio code » et tout mon travail a été placé sur un dépôt git [2] pour permettre à mes tuteurs de suivre mon travail facilement.

Contexte

Avant de développer le contenu du stage, il est nécessaire de parler de MUTE et du suivi des clients dans un système distribués pour aborder la problématique de ce stage.

MUTE

MUTE (Multi-Users Text Editing) [3] est un logiciel de recherche qui a été développé par l'équipe COAST. Il s'agit d'une application web qui, comme son nom l'indique, permet à plusieurs utilisateurs d'éditer un même document textuel depuis un navigateur web. Pour ce faire, les utilisateurs sont connectés en pair-à-pair, MUTE appartient au domaine des "systèmes distribués". Pour expliquer le fonctionnement de MUTE, on peut parler de 3 couches principales :

- Couche réseau : gestion de l'envoi et de la réception des messages entre clients

- Couche application : gestion des données et de leur réplication / gestion du comportement de l'application

- Couche interface : affichage et éditeur de texte

La prochaine section présentera plus précisément les notions de pair-à-pair et de systèmes distribués, mais nous allons tout d'abord exposer le problème qui justifie mon travail.

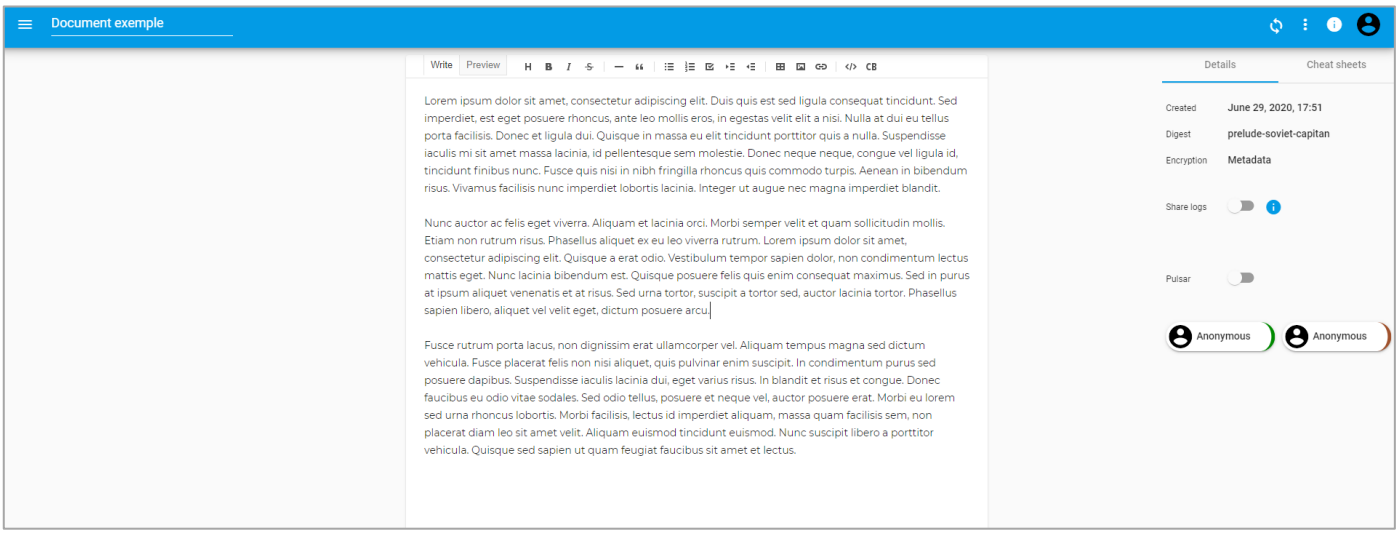


Figure 1 : Capture d'écran de l'interface de MUTE (https://coedit.re/)

Problème

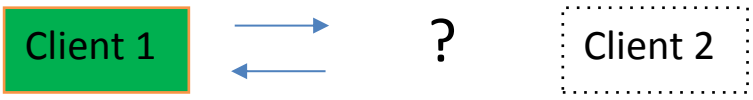
L'un des aspects importants des systèmes distribués est de savoir à tout moment qui fait partie du réseau. Le comportement d'un client varie souvent en fonction de celui des autres. Dans le cadre d'un logiciel où l'utilisateur n'intervient pas directement, la connaissance du réseau va permettre d'assigner des rôles aux clients, de répartir la charge de calcul ou les données à stocker. Mais il devient d'autant plus important de connaître les clients sur le réseau dans le cadre de l'édition collaborative de texte car le comportement des utilisateurs est fortement modifié s'ils sont plusieurs sur le même document. Chaque utilisateur va adapter son comportement en fonction de l'agissement des autres (chacun son paragraphe ; une personne écrit, une autre relit…). Cependant, comme nous allons le voir, il n'est pas simple de suivre l'évolution des états des clients dans le temps.

Tout le long de ce rapport, nous parlerons des clients comme étant *ok* ou *ko*. Un client *ok* est un client qui fonctionne normalement et qui doit rester sur le réseau et un client *ko* est un client qui est défaillant et qui doit être retiré du réseau.

Pour introduire le problème lié au suivi de l'évolution des états des clients, on considère la situation suivante :

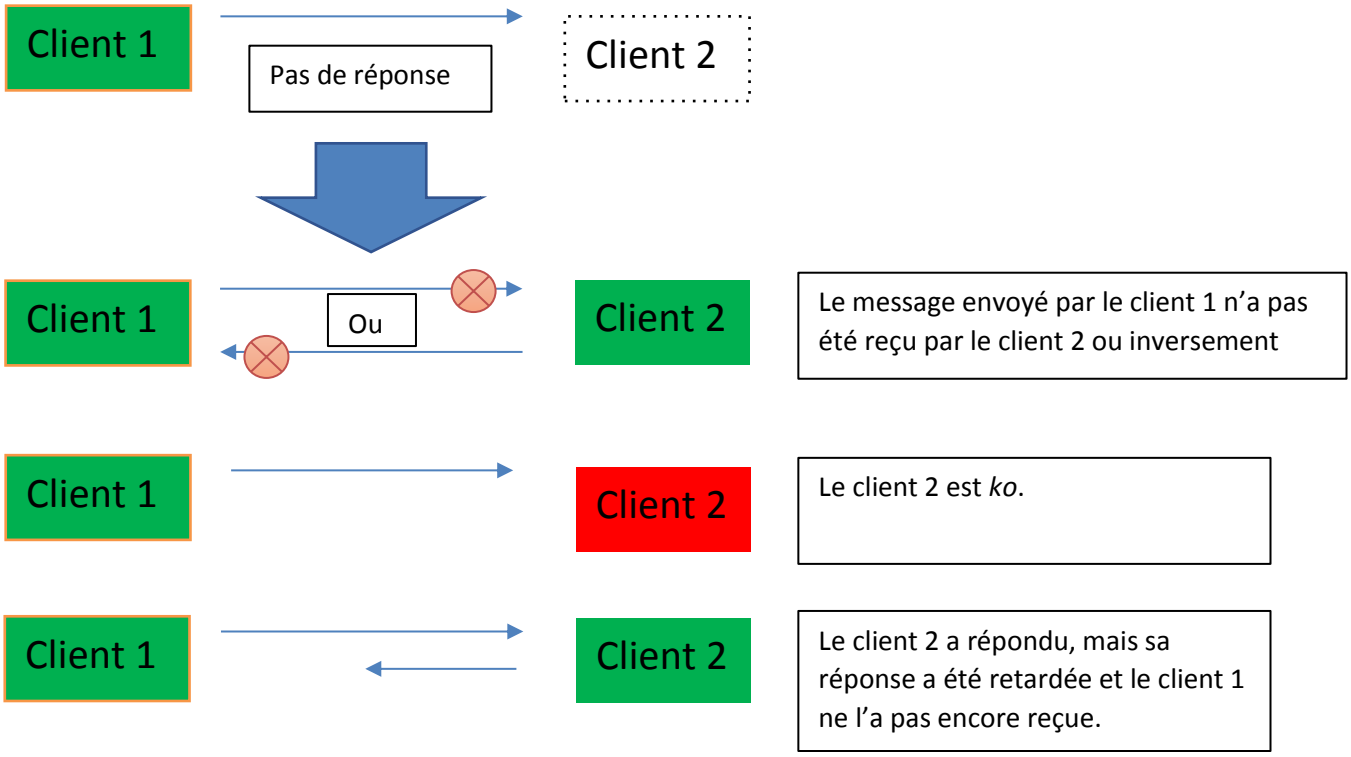
- 2 clients connectés en pair à pair communiquent.

- Cependant, chaque client connaît uniquement son état (ici *ok* en vert ou *ko* en rouge). Voici donc le point de vue du client 1 :



- Le client 1 sait uniquement qu'il envoie des messages au client 2 (sans savoir s'il les reçoit) et qu'il en reçoit du client 2.

Donc, si le client 1 ne reçoit plus de message du client 2, plusieurs cas sont possibles :



Cette situation met en évidence plusieurs choses :

1- Un client *ok* peut très bien être perçu par un autre comme *ko* ;
2- Un client qui ne répond pas à un message n'est pas forcément *ko*.

Cependant, dans la majorité des logiciels distribués, on ne peut pas se permettre de considérer un client *ko* alors qu'il ne l'est pas et il est important de détecter les clients *ko* pour ne plus les contacter inutilement. Cela est d'autant plus vrai dans MUTE car, comme évoqué plus tôt, la liste des collaborateurs permet aux utilisateurs de s'organiser pour collaborer.

Pour cela, on utilise des protocoles appelés protocoles de membership. Un protocole de membership est un protocole qui va permettre de suivre l'état des différents clients qui font partie du réseau.

Dans MUTE, cela est actuellement assuré par un protocole créé par l'équipe elle-même dont l'implémentation est couplée à la couche réseau. Cette couche réseau n'étant plus maintenue, deux problèmes se posent : tout d'abord, au fur et à mesures des mises à jour des navigateurs, le code génère de plus en plus de bugs car la technologie utilisée (WebRTC) a évolué rapidement. De plus,

ce couplage empêche aussi de changer de couche réseau, ce que l'équipe envisage de faire pour régler les problèmes liés à l'utilisation de WebRTC.

L'objectif du stage est justement d'implémenter le protocole de membership SWIM (sur lequel on reviendra dans la prochaine partie) dans MUTE.

Déroulement du stage

Le stage s'est donc déroulé en trois étapes : une partie de formation, une partie de développement d'un prototype et enfin une partie d'intégration dans MUTE (voir diagramme figure 2).

La partie de formation m'a permis d'appréhender les concepts nécessaires à la réalisation concrète du stage comme les protocoles de membership ou bien les flux RxJS.

Ensuite, le développement du prototype à consister en la programmation d'une application pour tester une implémentation du protocole SWIM. Cette partie a été très important car il a permis à la fois de produire plus facilement le code qu'il m'a ensuite fallu réutiliser lors de l'intégration, mais il a surtout un rôle de documentation auquel les membres de l'équipe peuvent se référer (s'ils s'intéressent au sujet de mon travail, ou tout simplement pour compléter l'intégration si besoin).

Enfin, l'intégration dans MUTE consiste à ajouter les fonctionnalités apportées par mon travail en s'assurant qu'il n'y a pas de régression sur le contenu existant.

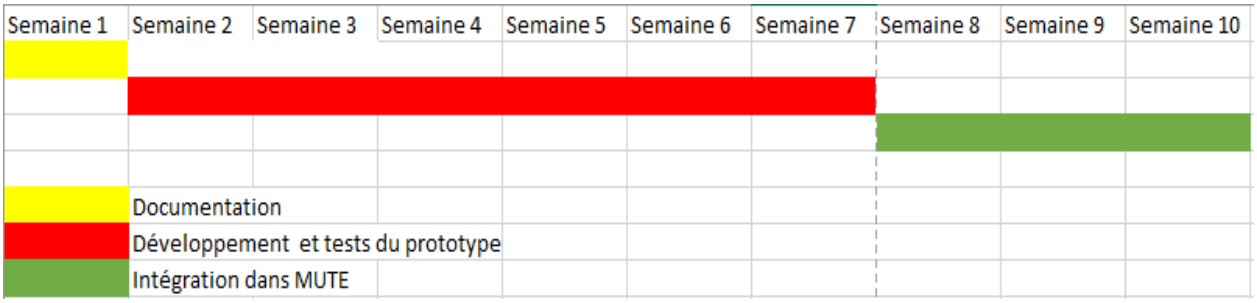


Figure 2 : Diagramme de GANTT du stage

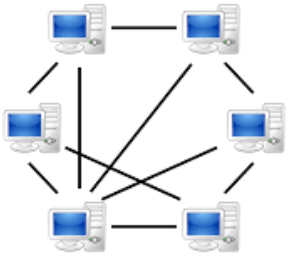
Etat de l'art

Durant la première semaine de stage, je me suis documenté sur les domaines que j'allais être amené à appréhender. Pour cela, mes tuteurs avait préparé plusieurs documents qui ont constitués un point de départ [4].

Nous allons donc, dans un premier temps, évoquer les bases à connaître pour entrer dans le sujet.

Pair-à-pair

Un système pair-à-pair est un système où les échanges entre clients sur le réseau se font par des échanges directs entre les clients. Ce type de système est à comparer aux systèmes client/serveur où tous les messages se font entre un client et le serveur. On dit que tous les nœuds du réseau sont à la fois client et serveur.



Systèmes distribués

La notion de système distribué correspond à la réalisation d'une tâche par plusieurs machines (ou « node » en anglais). Cette tâche peut aussi bien être de stocker des données, que de réaliser un calcul ou de faire fonctionner un logiciel.

Conflits de modifications et CRDT

L'une des grandes difficultés à gérer dans le cadre des systèmes distribués réside dans la notion de simultanéité des opérations. Comme les clients ne connaissent pas l'état global du système, ils agissent tous de manière indépendante. Il est donc possible qu'ils génèrent plusieurs opérations de manière concurrente. On parle de conflit de modification quand deux opérations simultanées se « contredisent ». Par exemple, il y a conflit quand deux opérations affectent des valeurs différentes à un même objet.

La gestion et la résolution d'un conflit peut être complexe. C'est pourquoi des structures de données ont été spécialement conçues pour éviter ou résoudre automatiquement les conflits de modification. Ces structures de données sont communément appelées CRDTs, « **C**onflict-free **R**eplicated **D**ata **T**ypes ».

Sans entrer dans les détails, cela signifie que certaines structures de données conçues pour être adaptées aux systèmes distribués ont été prouvées comme ne pouvant pas créer de conflit irrésoluble.

MUTE est justement l'implémentation d'un CRDT développé par l'équipe conçu pour l'édition collaborative de texte appelé LogootSplit.

Protocole de membership

Un protocole de membership est un protocole qui permet de suivre l'état des clients d'un réseau dans le cadre des systèmes distribués.

Pour bien comprendre, prenons l'exemple d'un protocole très simple, le « heartbeat protocol » : chaque client envoie périodiquement un message à tous les autres clients. Ce message, dénommé « heartbeat », permet à un client de signaler aux autres qu'il est toujours en ligne, dans l'état que nous avons précédemment nommé *ok*.

Si un premier client reçoit le « heartbeat » d'un deuxième client, alors le premier peut considérer avec assurance que le second est en ligne. A l'inverse s'il ne le reçoit pas depuis un certain temps déterminé, alors il peut considérer que le second client est hors ligne. Le protocole propose donc bien un mécanisme pour détecter la présence ou l'absence de clients dans la collaboration.

Ce protocole présente plusieurs problèmes :

1- Un client qui n'arrive pas à m'envoyer de message n'est pas forcément *ko* (connexion retardée, routage lent) ;
2- Tous les clients n'auront pas forcément la même liste de clients connectées ;
3- L'augmentation du nombre de messages sur le réseau augmente de manière quadratique.

On voit bien que ce protocole simple n'est pas adapté dans un contexte réel. Nous abordons dans la prochaine section le protocole de membership SWIM qui résout ces problèmes.

Le protocole SWIM

Le protocole SWIM [5] signifie « **S**calable **W**eakly-consistent **I**nfection-style Process Group **M**embership Protocol », ce qu'on peut traduire par « protocole de membership évolutif et faiblement cohérent d'adhésion à un groupe par infection ». Nous allons expliquer chacun des termes constitutifs de SWIM :

- Evolutif (scalable) : comme évoqué dans l'exemple du « heartbeat protocol », le comportement d'un protocole quand le nombre de clients augmente est très important. SWIM est défini comme évolutif car l'évolution du nombre de messages sur le réseau en fonction du nombre de pair est linéaire.
- Faible cohérence (*Weakly-consistent*) : la cohérence est un terme qui appartient au domaine des systèmes distribués est qui est relatif à l'état des données répliquées localement pour chaque client. Un système faiblement cohérent assure qu'au final, tous les clients auront les mêmes données même s'il est possible, qu'à un moment donné, deux clients ne soient pas dans le même état.
- Infection (Infection-style) : le terme infection est utilisé pour définir la manière dont les clients font circuler les informations sur le réseau. On parle d'infection car chaque client va transmettre une nouvelle information à quelques autres clients qui feront de même jusqu'à ce que tout le réseau ait perçu cette information, comme un virus peut se propager dans une population.

SWIM va donc permettre de suivre l'état des clients du réseau, mais aussi de répandre les informations concernant l'arrivée ou le départ d'un client (le départ d'un client étant légèrement différent de son échec).

Expliquons le fonctionnement de SWIM en détaillant les étapes d'une période (figure 3) :

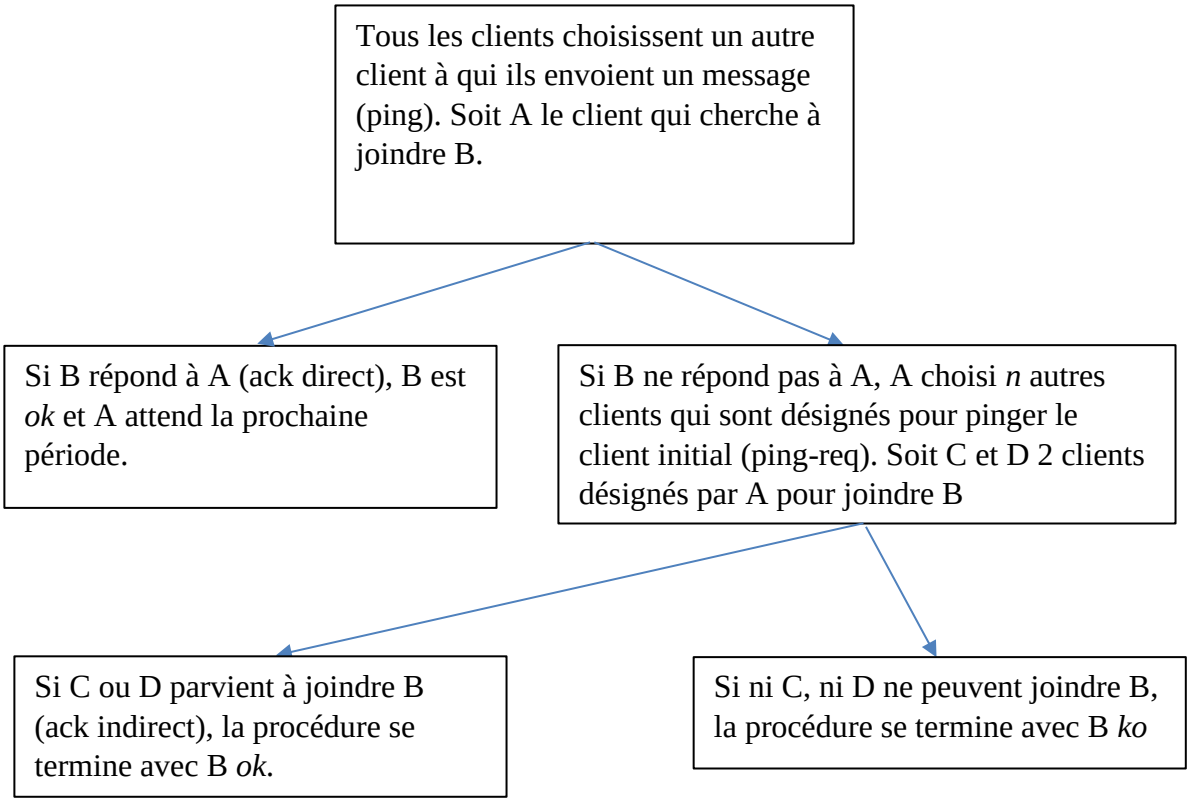


Figure 3 : Déroulement d'une période du protocole SWIM

A l'issu de cette période, certains clients peuvent être déclarés *ko*. Plutôt que de les exclure directement du réseau, SWIM propose un état intermédiaire où le client est « suspect ». Si un client suspect ne parvient pas à montrer qu'il est *ok*, il finit par être déclaré défaillant et est exclu du réseau. Ce mécanisme de suspicion va permettre de réduire grandement le risque d'exclusion d'un client *ok* mais va ralentir l'expulsion des clients réellement *ko*.

Pour faire parvenir à toutes les clients les informations relatives aux statuts des clients dans le réseau, SWIM réutilise les messages ping, ping-req et ack auxquels il rajoute les informations Alive, pour un client *ok*, Suspect pour un client suspect et Confirm pour un client *ko*.

Cette réutilisation (le terme exact est « piggyback ») permet de ne pas générer de messages additionnels et de garder un nombre bien déterminé de messages circulant sur ce réseau.

En résumé, on peut dire que le protocole SWIM doit permettre de suivre la liste des collaborateurs de manière fiable, simple et évolutive même si la durée pour détecter un client *ko* pourrait en être rallongée.

Prototype

Nous allons détailler ici le fonctionnement et la création du prototype qui a permis de tester notre implémentation du protocole SWIM.

Ce prototype a pour objectif de simuler une application collaborative temps réel en pair-à-pair. Nous avons décidé que chaque client devrait pouvoir exécuter 4 actions :

1. Ajouter un caractère dans un set (pour simuler les données d'une vraie application)
2. Bloquer les messages provenant d'un autre client (pour tester le comportement du protocole quand un client ne répond pas)
3. Déclencher un ping sur un autre collaborateur
4. Quitter le réseau

Il a connu 2 versions principales qui fournissent le même résultat bien que la conception de la deuxième soit meilleure. Il a été développé en TypeScript pour faciliter l'intégration dans le code existant, qui est lui aussi écrit dans ce langage. Nous étudions chaque version séparément.

Le code du prototype est accessible via la bibliographie [2] et une capture d'écran de l'interface du prototype est disponible en annexe 1.

Première version

Dans un premier temps, l'un des prérequis est de pouvoir connecter des clients entre eux. Pour cela, j'ai réutilisé du code disponible en ligne [6] qui permettait de connecter des clients dans un tchat et d'envoyer des messages. Ce code était construit autour de deux fichiers javascript (client.js et serveur.js) : la partie serveur n'a été que très peu modifiée, là où la partie client n'a gardé qu'une infime partie du code de base.

Cela s'explique par le fait que dans le cadre de MUTE, il n'y a pas de serveur. Ici, le serveur sert juste à connecter les clients entre eux et il se contente de propager à tous les clients connus les messages qu'il reçoit.

Avec cette implémentation, on a un client par onglet d'ouvert en local.

Par exemple, imaginons que le client 2 veut « pinger » le client 3. On considère alors le schéma ci-dessous.

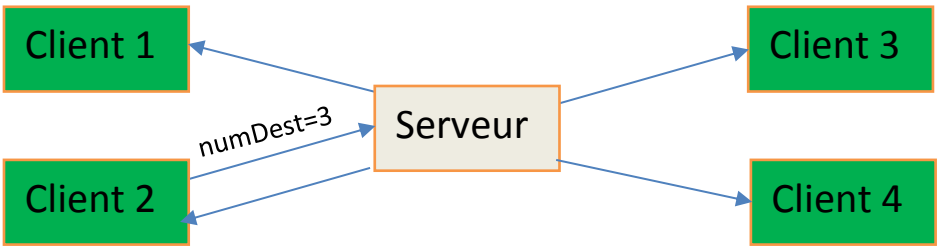


Figure 4 : Schématisation du routage d'un message du prototype

Le client 2 s'est contenté d'envoyer un message au serveur qui contient l'information du numéro du destinataire et le serveur a broadcasté cette information à tous ses clients. C'est ensuite au client de filtrer les messages pour garder uniquement ceux qui le concerne et retirer les messages qui doivent être bloqués.

Pour simuler le fonctionnement d'une vraie application, chaque client peut ajouter des caractères à un set. Pour valider le prototype, il sera important de vérifier que tous les clients aient le même set si on laisse le temps à la situation de se stabiliser.

On a ensuite commencé à suivre la liste des collaborateurs. Dans un premier temps, chaque utilisateur conserve en mémoire tous les utilisateurs connus et seul un utilisateur qui quitte volontairement la page peut être supprimé de la liste des autres collaborateurs. Cela est assez simple à implémenter, et comme cet état n'est que transitoire, nous n'allons que l'évoquer.

Enfin, l'implémentation de SWIM va permettre de vérifier la présence des collaborateurs et de retirer ceux qui sont *ko*.

L'annexe 2 contient tous les messages qui peuvent transiter sur le réseau et toutes les informations possiblement « piggybackées » sur un message.

Ci-dessous, nous présentons les différents champs que doivent contenir un message :

- Nous avons déjà parlé du numéro du destinataire (numDest) ;
- Il faut aussi ajouter le numéro de l'envoyeur pour permettre une réponse (numEnvoi) ;
- Il faut bien sûr préciser quel est le message envoyé (type). Comme il n'existe que peu de messages différents, on enverra plutôt un numéro qu'une chaine de caractère dans le réseau ;
- Il faut également envoyer les informations relatives aux statuts des clients qui doivent être « piggybacked» (piggyback). Pour cela, tous les messages doivent contenir un tableau (qui peut être vide) qui contiennent ces informations.
- Enfin, à chaque message, les clients échangent leur set pour garder l'union des deux en cas de différences (set).

Ces 5 informations sont essentielles et présentes dans tous les messages propres à SWIM. Cependant, d'autres informations peuvent être présentes sur le réseau :

- Dans le cas d'un ping-req, il faut préciser le numéro du collaborateur à ping (numCible) ;
- Dans le cas d'un ping-reqRep, il faut préciser si la cible a répondu au ping ou non (reponse) ;
- Dans le cas d'un data-update, on envoie également des données qui sont censées rester locales (l'état actuel de la liste des collaborateurs).

Pour donner un exemple concret, la figure 5 montre l'interface utilisée pour définir un message ping dans le code du prototype.

On retrouve nos 5 informations obligatoires avec les champs : type, numEnvoi, numDest, set et piggyback.

```
export const TYPE_PING_LABEL = 'ping';
export interface Ping {
    type: typeof TYPE_PING_LABEL;
    numEnvoi : number;
    numDest : number;
    set : string[];
    piggyback: [number,MessPG][];
}
```

Figure 5 : Interface qui représente un message 'ping'

On peut aussi schématiser un échange de message possible sur le réseau. On représente ici avec la figure 6 l'une des situations les plus simple, c'est-à-dire un échange ping/ack.

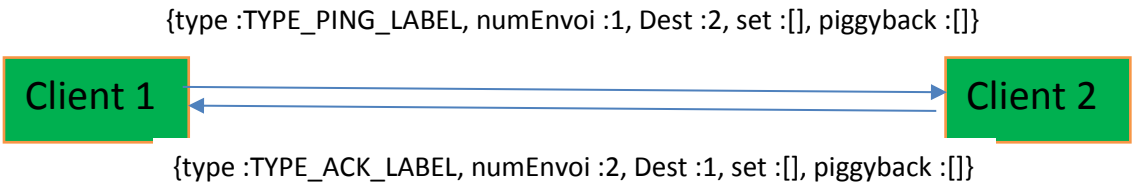


Figure 6 : Représentation d'un message ping et de sa réponse ack

Une fois tous les messages créés, on déclare quel doit être la réaction à un message donné (par exemple, un ping venant de x doit générer un ack vers x). Globalement, on peut résumer ces comportements avec ces trois relations :

- Ping provoque ack.
- Ping-req provoque ping puis ping-reqRep
- Data-request provoque data-update

Pour finir, on programme une méthode qui sera exécutée périodiquement (une période dure environ 2 secondes) et qui réalisera globalement le schéma détaillé dans la partie sur SWIM (figure 3).

Deuxième version

Comme évoqué dans l'introduction, les deux versions produisent le même résultat (même si la deuxième version a été beaucoup plus testée à l'aide de tests unitaires, nous y reviendrons à la fin de cette sous-partie) mais la deuxième version se base sur une meilleure conception.

Conception

Cette conception dissocie trois éléments logiciels : la partie interface utilisateur (ui) qui gère l'affichage, la partie application (app) qui gère le protocole SWIM et la partie réseau (res) qui gère la réception et l'envoi des messages (notamment le blocage).

On considère le schéma ci-dessous (figure 7) :

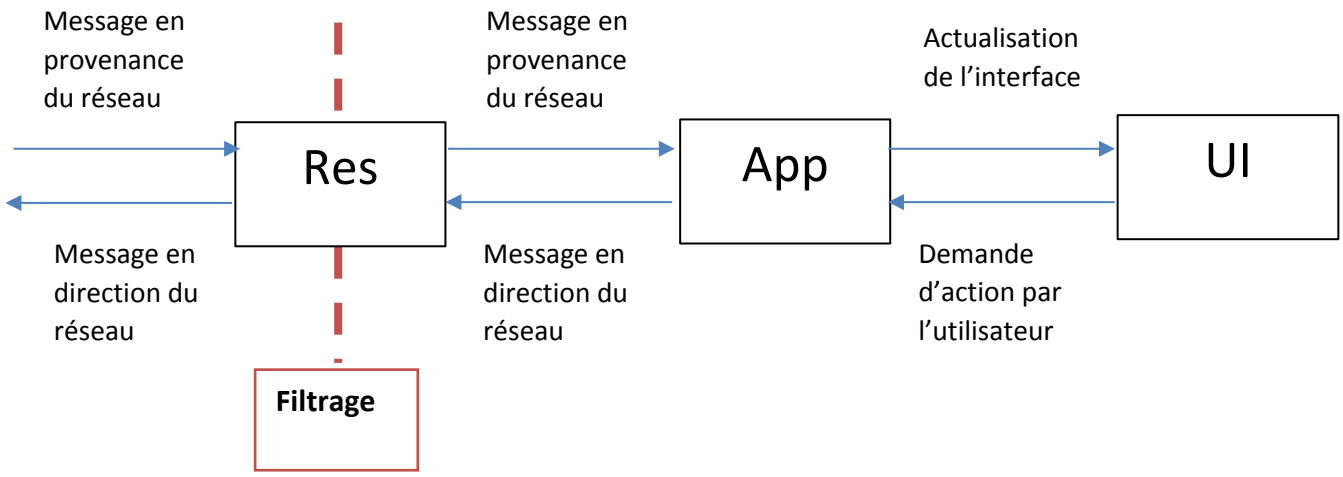


Figure 7 : Schématisation des interactions entre les composants logiciels du prototype

Chaque composant communique avec les autres à l'aide de message. Ils ne sont plus imbriqués les uns aux autres comme dans la première version. La communication entre les composants se fait à l'aide de Rxjs dont nous allons détailler le fonctionnement.

RxJS

RxJS [7] est une bibliothèque qui va permettre une « programmation réactive ». La programmation réactive est un paradigme de programmation qui se base sur la programmation asynchrone pour gérer les flux de données et la propagation des changements. La notion de flux de données est très importante car tous les objets RxJS sont des « flux » (ou Streams) qui émettent des événements au fil du temps. Mon utilisation de RxJS dans le cadre du stage se base sur deux concepts principaux : les sujets et les observables

Un sujet est un flux RxJS qui va émettre des événements de manière dynamique, et un observable est un flux qui peut être construit à partir d'un sujet auquel on va pouvoir s'abonner pour exécuter une fonction à chaque ajout.

Pour bien comprendre, on représente l'utilisation des sujets et observables dans le schéma suivant (figure 8) :

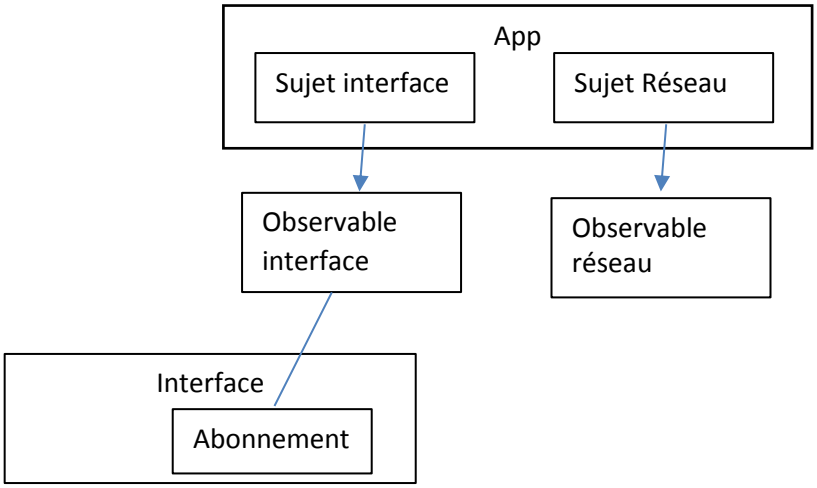


Figure 8 : Schéma résumant l'abonnement à un Observable RxJS

La couche applicative émet des événements dans le sujet interface et l'interface graphique les reçoit à travers l'observable interface. L'interface graphique est ainsi abonnée aux événements émis par la couche applicative. Pour autant, la couche applicative et l'interface graphique ne se connaissent pas directement. Cette architecture permet donc un découplage des deux entités.

Tests unitaires

Enfin, cette version a permis le développement de tests unitaires portant uniquement sur la couche applicative (app). Seul cette couche a été testée car il s'agit de la seule à être intégrée à MUTE. Comme le schéma le suggère (figure 8), toutes les informations externes à app vont lui parvenir à l'aide des observables des autres composants. En créant un sujet qui contient des valeurs prédéfinies, il est plus simple de vérifier la réaction de app qu'en fabriquant un scénario complet qui agit sur tous les composants.

On peut regrouper les fonctionnalités testées en 3 catégories :

1 - **Vérification du comportement d'un nœud lors de la réception d'un message** : les propriétés évoquées concernant le comportement attendu après un message d'un certain type sont respectées (exemple : après la réception d'un ping, je réponds un ack), les informations de piggyback sont propagées correctement.

2 - **Vérification des conséquences après l'exécution d'une période** : un client qui répond à un ping (par un ack direct ou indirect) doit rester *ok*, sinon il doit passer *ko* (s'il était Alive, il devient Suspect et s'il était Suspect il doit être retiré du réseau).

3 - **Vérification des mises à jour des structures de données** : ajout, suspicion ou suppression d'un collaborateur principalement.

Intégration dans MUTE

Nous allons maintenant aborder la dernière partie du stage : l'intégration du protocole SWIM à MUTE. Pour simplifier le contenu de cette partie, nous ne détaillerons pas le fonctionnement interne de l'application. On précise tout de même que ces modifications seront effectuées dans le composant appelé mute-core.

Mute-core

Mute-core est le composant de MUTE qui gère les données qui représentent le document textuel et les métadonnées qui l'accompagne. C'est donc à ce composant que nous avons attribué la responsabilité de maintenir la liste des collaborateurs. Contrairement au prototype, il a donc fallu tenir compte du code existant pour effectuer des mises à jour, ce qui a beaucoup complexifié l'implémentation. Il est d'ailleurs important de noter que la personne qui à réaliser cette partie du code ne fait plus partie de l'équipe ce qui a n'a pas facilité la compréhension du code existant.

De plus, comme évoqué dans l'introduction, la gestion des collaborateurs était à la base couplée à la couche réseau, il a donc fallu lors de l'implémentation séparer clairement le protocole SWIM du code gérant la couche réseau.

Tout cela a donc impliqué des tests fréquents de l'application complète pour vérifier que le nouveau code ne générait pas d'effets de bord sur le reste de l'application.

Différences entre le prototype et l'implémentation dans mute-core

Pour différentes raisons, des modifications entre le prototype et l'implémentation dans mute-core ont dues être faites :

- Données relatives à un collaborateur

Contrairement à ce qui a été fait dans le prototype, un collaborateur correspond à plus d'informations qu'un simple numéro. On y associe par exemple un pseudo, une adresse e-mail ou d'autres informations mais surtout un collaborateur possède plusieurs identifiants : un identifiant réseau, un identifiant mute-core et un identifiant pour l'appareil.

- Initialisation d'un collaborateur

Une autre différence réside dans la manière où un client rejoint le réseau. Dans le prototype, c'est le serveur qui crée la requête du nouveau client à destination des clients qui sont déjà dans le réseau. Dans mute-core, comme il n'y a pas de serveur, c'est le client qui est chargé d'envoyer une requête. Pour cela, il en envoie périodiquement jusqu'à obtenir une réponse.

- Utilisation de protobuf [8] pour les messages

Pour limiter la taille des messages envoyés sur le réseau, il est nécessaire de les encoder. Pour cela, nous avons utilisé la bibliothèque protobuf. Cela est relativement simple pour des éléments comme des nombres ou des chaînes de caractère, mais devient beaucoup plus difficiles avec des éléments complexes comme par exemple les tableaux associatifs.

Retours sur l'intégration de SWIM

Bien qu'il reste encore 1 semaine pour continuer l'intégration au moment de rendre ce rapport, on peut tout de même effectuer quelques retours sur cette intégration.

L'un des objectifs essentiels de cette implémentation était de dissocier la couche réseau et le suivi des collaborateurs. Dans le cadre de la gestion des collaborateurs, la couche réseau était utilisée pour 2 usages :

- Détecter les arrivées et départs sur le réseau
- Router et envoyer des messages.

Grâce à l'implémentation effectuée, on a bien supprimé la partie du code qui détectait les nouvelles connexions pour ajouter des collaborateurs puis les retirer une fois la connexion fermée.

Cependant, j'ai déjà évoqué qu'il existait plusieurs identifiants pour un collaborateur. C'est l'identifiant réseau qui est utilisé dans mute-core pour router les messages au lieu de l'identifiant mute-core lui-même. C'est une incohérence qui est présente dans le code existant et qui découle de sa conception. Ce défaut de conception est trop important pour être corrigé en quelques jours. Nous avons donc choisi d'ignorer ce problème pour le moment. Nous utilisons donc toujours les identifiants réseaux pour router les messages du protocole à l'heure actuelle. Pour résoudre ce problème et découpler la couche réseau du suivi des collaborateurs, il faudrait remplacer l'identifiant réseau par celui de mute-core dans le code qui concerne la gestion des collaborateurs. Il faudrait aussi mettre en place une table de correspondance entre les identifiants réseaux et mute-core.

Améliorations possibles

Avant de conclure, on peut lister quelques améliorations envisageables pour améliorer le fonctionnement ou la conception de l'application.

- Communiquer l'information "noeud suspect" à l'interface

Pour l'instant, le statut « Alive » ou « Suspect » est propre à la partie application et n'est pas utilisée pour mettre à jour l'interface. L'utilisation de cette information par l'interface pourrait servir à montrer aux utilisateurs quel collaborateur a une connexion instable (via par exemple un code couleur sur l'affichage des collaborateurs).

- Implémentation du « Round-Robin » pour choisir quel noeud pinger

Au lieu de pinger aléatoirement un autre client à chaque période, le « Round-Robin » est une solution où chaque client a sa propre liste ordonnée de manière aléatoire et va pinger les autres clients en suivant cette liste. Cela va permettre de borner avec certitude le temps nécessaire avant d'avoir transmis les dernières informations à un client particulier.

- Intégration des améliorations apportées par Lifeguard [9]

Lifeguard est un ensemble d'extensions applicable à SWIM pour améliorer certains facteurs comme le temps de détection d'un membre *ko* ou le taux d'expulsion injustifiée. Dans notre cas, le temps de détection n'impacte que très peu l'application (pour un utilisateur, une différence d'une seconde n'est pas critique). On peut donc se contenter d'un délai de détection plus long ce qui va aussi réduire le taux d'expulsion injustifiée. Ces extensions seraient donc utiles si de nouvelles fonctionnalités ont besoin d'un temps de détection plus court, mais dans l'état actuel de l'application, cela reste dispensable.

- Remplacer l'utilisation de l'identifiant réseau par l'identifiant mute-core pour le code qui ne sert pas directement à envoyer un message pour découpler totalement la couche réseau de la gestion de la liste des collaborateurs

Conclusion

En résumé, on peut dire que l'implémentation du protocole SWIM s'est relativement bien déroulée.

Pour ce qui est du prototype, les objectifs sont atteints car l'implémentation proposée répond aux attentes. En effet, les tests unitaires et les scénarios imaginés pour tenter de mettre en défaut l'algorithme ne créent pas d'états incohérents. Cela m'a permis de passer sereinement à la phase suivante.

C'est lors de l'intégration que j'ai rencontré le plus de problèmes car j'étais bien plus dépendant de l'aide de l'équipe. Certains problèmes que j'ai pu rencontrer provenaient de portion de code qui venait d'autres fichiers de plusieurs centaines de lignes ou bien même d'autres dépôts git. Au moment de la rédaction du rapport, la partie intégration est en cours. Au vu des problèmes rencontrés jusqu'à présent, je ne suis pas sûr de pouvoir finir l'intégration d'ici la fin de mon stage. Cependant le prototype étant complètement réalisé, documenté et testé, l'équipe aura toutes les ressources nécessaires pour finaliser l'intégration si besoin.

Sur un plan plus personnel, ce stage m'a beaucoup appris sur le monde de la recherche en général et sur les systèmes. Les conditions imposées par la crise actuelle m'ont permis de découvrir le télétravail et d'adapter mes méthodes de travail.

Bibliographie

[1] Site internet du LORIA : https://www.loria.fr/fr/

[2] Mon dépôt git : https://github.com/TomMendez/Stage_LORIA_MendezTom_2020

[3] MUTE : lien github : https://github.com/coast-team/mute lien application : https://coedit.re/

[4] Documents étudiés lors de la phase de formation : https://github.com/MatthieuNICOLAS/2020-stage-membership/

[5] SWIM : http://www.cs.cornell.edu/info/projects/spinglass/public_pdfs/SWIM.pdf

[6] Base du code utilisé pour créer le prototype (licence ISC) : https://github.com/markbrown4/websocket-demo

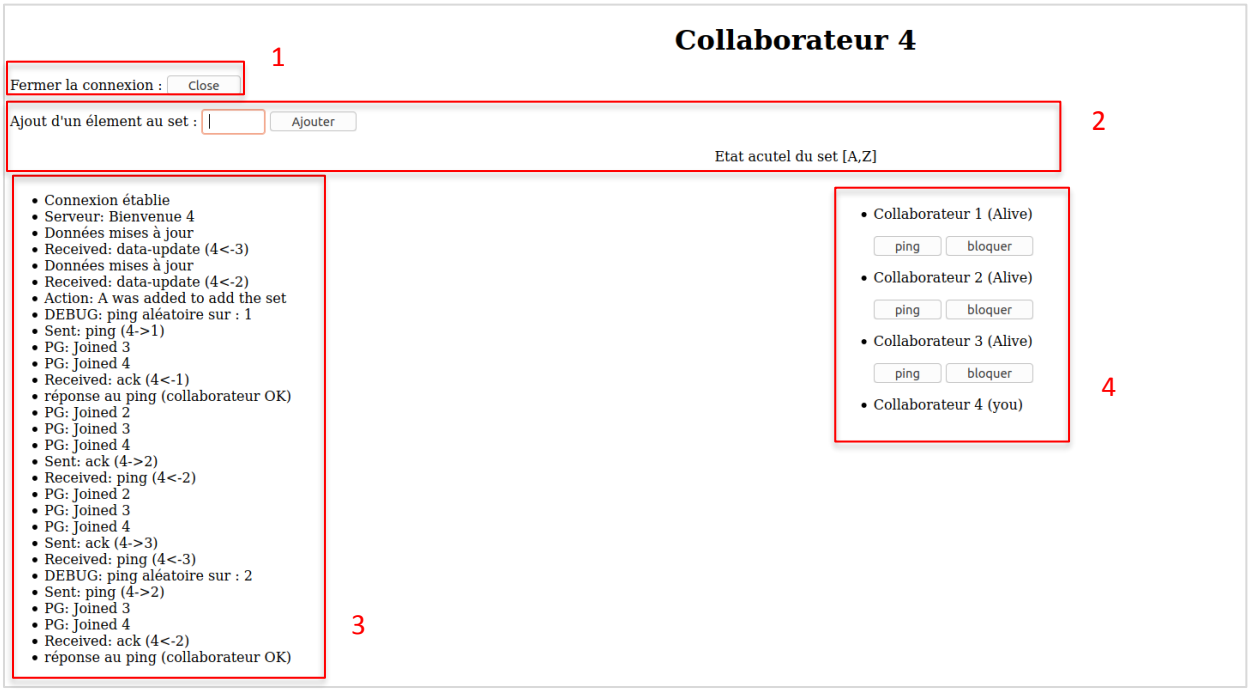
[7] RXJS : https://github.com/ReactiveX/RxJS ; https://www.learnrxjs.io/

[8] protobuf : https://github.com/protobufjs/protobuf.js ; https://developers.google.com/protocol-buffers

[9] Lifeguard : https://arxiv.org/abs/1707.00788

Dépôt git de l'équipe COAST : https://github.com/coast-team

Annexe 1 : Capture d'écran de l'interface du prototype



1 – Bouton pour fermer la connexion et quitter le réseau

2 – Etat du set et champ pour ajouter un caractère

3 – Log des événements

4 – Liste des collaborateurs : on voit les collaborateurs toujours en ligne et leur statut (Alive ou Suspect). Il est possible de bloquer un collaborateur pour simuler un problème sur le réseau ou de déclencher un ping.

Annexe 2 : Messages

Messages envoyés sur le réseau

Ping	Message basique qui doit provoquer un ack
Ping-req	Message envoyé pour demander à une machine de pinger une autre machine.
Ack	Réponse générée après réception d'un ping
Data-request	Message généré par le réseau pour une machine qui vient de rejoindre le réseau.
Data-update	Message généré après réception d'un data-request, il porte toutes les données du client (liste des collaborateurs & set)
Ping-reqRep	Message généré après un délai prédéfini qui suit la réception d'un ping-req pour informer l'état du ping envoyé à la machine ciblée.

Messages transportés en « piggyback »

Joined	Informe de l'arrivée sur le réseau d'une nouvelle machine
Alive	Message générée par une machine qui apprend qu'elle est suspectée pour redevenir une machine « Alive » c'est-à-dire qui est fonctionnelle.
Suspect	Message généré quand une machine ne parvient pas à en joindre une autre qu'il connait comme « Alive » par ping et par ping-req.
Confirm	Message généré quand une machine ne parvient pas à en joindre une autre qu'il connait comme « Suspect » par ping et par ping-req. Confirm va supprimer le client de la liste des collaborateurs, les message Alive et Suspect seront ignorés pour un client qui a été supprimé.

Annexe 3 : Fiche pour dépôt du mémoire à la bibliothèque

Fiche pour dépôt du mémoire de stage à la bibliothèque

Note :

(à remplir par le jury)

Nom et prénom de l'étudiant :

Mendez-Porcel Tom

Accord pour la diffusion du mémoire et son impression:

☒ oui ☐ non

Date : 25/06/2020

Signature de l'étudiant :

DUT : ☒ Deuxième année ☐ Année spéciale

☒ Informatique

Licence pro :

☐ ASRALL
☐ CIASIE
☐ Autre :

Titre du mémoire :

Implémentation du protocole de membership SWIM à un logiciel d'édition collaborative de texte

Tuteur : Matthieu Nicolas

Nom de l'entreprise : LORIA

Adresse : 2 ter boulevard Charlemagne, Nancy

Type d'activité (domaines couverts par l'entreprise) :

Mots-clés (sujets traités) : Systèmes distribués ; systèmes pair-à-pair ; protocole de membership ; SWIM

Résumé :

L'objectif de mon stage a été d'ajouter une fonctionnalité au logiciel MUTE (application web de traitement de texte en pair-à-pair) qui consiste à suivre les collaborateurs connectés à l'application.