Page de couverture



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 lors de ce stage. Sans leurs explications, conseils et retours fréquents sur mon travail, je n’aurais pas pu réaliser le quart de mon travail.

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 si j’avais pu me rendre au laboratoire pour ce stage tout comme le reste du personnel sur place.

Enfin, j’aimerais remercier 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 mon stage.

Sommaire

[Introduction 4](#_Toc43185171)

[Présentation du LORIA 4](#_Toc43185172)

[Présentation de l’équipe COAST 4](#_Toc43185173)

[Cadre de travail 5](#_Toc43185174)

[Contexte 6](#_Toc43185175)

[MUTE 6](#_Toc43185176)

[Problème 6](#_Toc43185177)

[Etat de l’art 8](#_Toc43185178)

[Pair à pair 8](#_Toc43185179)

[Systèmes distribués 8](#_Toc43185180)

[Conflits et CRDT 8](#_Toc43185181)

[Protocole de membership 9](#_Toc43185182)

[Le protocole SWIM 10](#_Toc43185183)

[Prototype 12](#_Toc43185184)

[Première version 12](#_Toc43185185)

[Deuxième version 14](#_Toc43185186)

[Conception 14](#_Toc43185187)

[RxJS 15](#_Toc43185188)

[Tests unitaires 16](#_Toc43185189)

[Intégration dans MUTE 17](#_Toc43185190)

[Conclusion 18](#_Toc43185191)

[Bibliographie 19](#_Toc43185192)

[Annexe 1 : Capture d’écran de l’interface du prototype 20](#_Toc43185193)

[Annexe 2 : Messages 21](#_Toc43185194)

# Introduction

## 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 reparties 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*

## 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.

Nous nous intéressons 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 mon tuteur. J’ai donc eu la chance de pouvoir travailler dans des conditions convenables aussi bien au niveau du matériel que de l’atmosphère de travail.

Pour la réalisation du code, j’ai utilisé le logiciel « visual studio code » et tout mon travail à été placé sur un dépôt git pour permettre à mon tuteur de suivre mon travail facilement.

L’objectif de mon stage a été d’ajouter une fonctionnalité au logiciel MUTE qui consiste à suivre les collaborateurs connectés à l’application. On reviendra plus en détail sur le sujet dès le début de la prochaine partie, nous ferons ensuite quelques explications théoriques nécessaires avant de passer en revue chacune des 3 phases du stage chronologiquement.

# Contexte

## MUTE

MUTE (Multi-Users Text Editing) est un logiciel de recherche qui a été développé par l’équipe COAST. C’est une application web qui, comme son nom l’indique, permet à plusieurs utilisateurs d’éditer un même document texte depuis un navigateur. Pour cela, tous les utilisateurs sont connectés en pair à pair, MUTE rentre donc dans le domaine des “systèmes distribués”. Nous reviendrons plus en profondeur dans la prochaine section sur les notions de pair à pair et de systèmes distribués, mais nous allons tout d’abord exhiber le problème qui justifie mon stage.

## Problème

Pour introduire ce problème, on considère la situation suivante :

* 2 clients connectés (en pair à pair) communiquent

Client 2

Client 1

* Cependant, chaque client connait uniquement son état (ici ok en vert ou ko en rouge) donc voici le point de vue du client 1

?

Client 2

Client 1

* Ce qui est schématisé, c’est que 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 messages du client 2, plusieurs cas sont possibles :

Client 2

Pas de réponse

Client 1

Le message envoyé par le client 1 n’a pas été reçu par le client 2 ou inversement

Client 1

Client 2

Ou

Le client 2 est ko

Client 1

Client 2

Client 1

Le client 2 a répondu, mais sa réponse a été retardée et le client 1 ne l’a pas encore reçu

Client 2

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 à un message n’est pas forcément ko

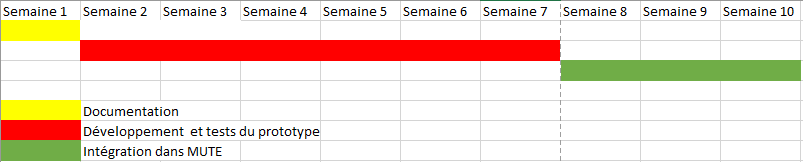
Cependant, dans la majorité des logiciels, on ne peut pas se permettre de considérer un client ko alors qui ne l’est pas et il est important de détecter les clients ko pour ne plus les contacter inutilement (et MUTE ne fait pas exception).

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 (quel qu’il soit).

Dans MUTE, cela est actuellement assuré par un protocole créé par l’équipe elle-même dont l’implémentation est couplée à la couche réseau. Cela pose un double problème car cette couche réseau n’est plus maintenue et l’application à de plus en plus de mal à suivre correctement la liste des collaborateurs. Mais ce couplage empêche aussi de changer de couche réseau.

L’objectif du stage est justement d’implémenter le protocole de membership SWIM (sur lequel ont reviendras dans la prochaine partie) à MUTE.

Ce stage a donc été séparé en 3 parties : une partie de documentation pour appréhender les concepts, une partie de développement d’un prototype et enfin une partie d’intégration dans MUTE (cf planning ci-dessous).

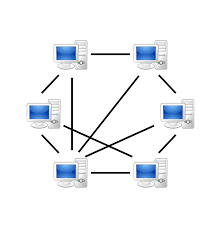


# Etat de l’art

Durant la première semaine de stage, je me suis uniquement documenté sur les domaines que j’allais être amené à appréhender pour mon stage.

Pour cela, mon tuteur avait préparé plusieurs documents sur lesquels je pouvais me baser (cf [bibliographie](#_Bibliographie))

Nous allons donc dans un premier temps évoquer les bases à connaitre pour rentrer 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 (contrairement à un système 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, de réaliser un calcul ou de faire fonctionner un logiciel.

## Conflits et CRDT

Cependant, le fait que plusieurs opérations puissent arriver simultanément apporte de nouveau problème. On parle de conflit quand 2 opérations simultanées se « contredisent ». Pour être plus précis, il y a conflit quand 2 opérations affectent la même propriété d’un même objet.

La gestion et la résolution d’un conflit peut être complexe, cependant, dans notre cas, nous sommes dans une situation de CRDT.

CRDT signifie « Conflict-free replicated data type », nous ne rentrerons pas dans les détails, mais certaines structures de données ont été prouvées comme ne pouvant pas créer de conflit irrésoluble. En admettant que MUTE en fait partie (car il existe des algorithmes permettant de « gérer et résoudre » les conflits automatiquement pour les éditeurs de texte), on peut commencer à créer l’implémentation qui permettras de gérer automatiquement la liste de collaborateur.

## Protocole de membership

Comme évoqué dans la partie précédente, 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 machine va envoyer à chaque période T un message à toutes les autres uniquement pour les notifier que cette machine est toujours ok.

Ce protocole permet donc bien de suivre l’état de toutes les machines du réseau :

* Les machines qui m’ont envoyé un message depuis T sont ok toutes les autres sont ko

Cependant, ce protocole comporte plusieurs problèmes :

1. Comme évoqué plus tard, une machine 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 machines connectées
3. L’augmentation du nombre de messages sur le réseau augmente de manière quadratique

On voit donc bien que ce protocole basique ne soit pas vraiment adapté pour une implémentation réelle. Nous allons donc maintenant parler du protocole SWIM et mettre en évidence quelles sont les solutions à ces problèmes.

## Le protocole SWIM

Parlons maintenant du protocole SWIM. SWIM signifie « **S**calable **W**eakly-consistent **I**nfection-style Process Group **M**embership Protocol », ce qu’on peut traduire en français par « protocole de membership évolutif et faiblement cohérent d’adhésion à un groupe par infection ». Nous allons expliquer chacun des termes que cachent 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 message sur le réseau en fonction du nombre de pair est linéaire (à vérifier).
* 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 2 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 machines du réseau, mais aussi de répandre les informations concernant l’arrivée ou le départ d’une machine (le départ d’une machine étant légèrement différent de son échec).

Nous allons maintenant expliquer le fonctionnement de SWIM en détaillant les étapes d’une période :

Si B répond à A (ack direct), B est ok et A attends la prochaine période.

Toutes les machines choisissent une autre machine à qui elles envoient un message (ping). Soit A la machine qui cherche à joindre B.

Si B ne répond pas à A, A choisi *n* autres machines qui sont désignées pour pinger la machine initiale (ping-req). Soit C et D 2 machines désignées par A pour joindre B

Si C ou D parvient à joindre B (ack indirect), la procédure se termine avec B ok.

Si ni C, ni D ne peuvent joindre B, la procédure se termine avec B ko

A l’issu de cette période, certaines machines peuvent être déclarées ko. Plutôt que de les exclure directement du réseau, SWIM propose un état intermédiaire ou la machine est « suspecte ». Si une machine suspecte ne parvient pas à montrer qu’elle est ok, elle finit par être déclarée défaillante et est exclue du réseau.

Pour faire parvenir à toutes les machines 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 une machine ok, Suspect pour une machine suspecte et Confirm pour une machine ko.

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

# Prototype

Nous allons maintenant détailler le fonctionnement et la création du prototype qui a permis de tester notre implémentation du protocole SWIM. Ce prototype a connu 2 versions principales qui fournissent le même résultat bien que la conception de la deuxième soit bien meilleure. Nous allons détailler chaque version séparément.

On note qu’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 (cf [bibliographie](#_Bibliographie)) qui permettait de connecter des clients dans un tchat et d’envoyer des messages. Ce code était construit autour de 2 fichiers javascript (client.js et serveur.js) et la partie serveur n’a été que très peu été 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 localhost.

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

Client 3

Client 1

numDest=3

Serveur

Client 4

Client 2

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.

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.

On réfléchit aux informations 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 envoyer (message). Comme il n’existe que peu de messages différents, on enverra plutôt un numéro qu’une chaine de caractère dans le réseau.
* Il faut aussi envoyer les informations relatives aux statuts des clients qui doivent être « piggybacked ». Pour cela, tous les messages doivent contenir un tableau (qui peut être vide) qui contiennent ces informations.

Ces 4 informations sont essentielles et présentes dans tous les messages propres à SWIM. Cependant 3 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 à répondu au ping ou non (reponse)
* Enfin, à chaque message, les clients échangent leur set pour garder l’union des 2 en cas de différences (set)

Exemple d’échange possible sur le réseau (qui correspond au cas le plus simple ping/ack) :

{message :1, numEnvoi :1, Dest :2, set :[], piggyback :[]}

Client 2

Client 1

{message :3, numEnvoi :2, Dest :1, set :[], piggyback :[]}

Une fois tous les messages créés, il faut déclarer quel doit à 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 3 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.

## Deuxième version

Comme évoqué dans l’introduction, les 2 version produisent le même résultat (même si la deuxième version a été beaucoup plus testée à l’aide de tests unitaires, nous y reviendront à la fin de cette sous-partie) mais la deuxième version se base sur une bien meilleure conception.

### Conception

On considère le schéma ci-dessous :

Application

Réseau

Message en provenance du réseau

Message en provenance du réseau

Actualisation de l’interface

Interface

**Filtrage**

Demande d’action par l’utilisateur

Message en direction du réseau

Message en direction du réseau

Cette conception dissocie 3 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).

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

### RxJS

Rxjs est une librairie qui va permettre une « programmation reactive ». La programmation réactive est un paradigme de programmation qui se base sur la programmation asynchrone pour gérer les flux de données et la propagation du changement. Ici, on va utiliser des sujets et des observables Rxjs.

Un sujet est un objet Rxjs qui va recevoir des objets de manière dynamique, et un observable est un objet 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 :

App

Sujet interface

Sujet Réseau

Observable interface

Observable réseau

Interface

Abonnement

L’objectif de ce schéma, c’est de montrer que le réseau s’abonne à l’observable bien qu’il ne connaisse pas directement le sujet.

### Tests unitaires

Enfin, cette version va aussi permettre le développement de tests unitaires portant uniquement sur la partie app (qui sera la seule à être intégrée à MUTE). Comme vu sur le schéma, 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 fabricant un scénario complet qui agit sur tous les composants.

# Intégration dans MUTE

?

# Conclusion

Au moment de la rédaction du rapport, la partie intégration n’a qu’été commencée donc cette conclusion portera plutôt sur le reste du stage. De toute manière, si la réalisation d’un prototype est réalisée correctement avec documentation et tests, l’intégration pourrait être réalisée par quelqu’un d’autre bien plus facilement que si l’intégration est commencée mais qu’il n’existe aucun test et peu de documentation.

Pour ce qui est du prototype donc, on peut dire que les objectifs sont atteints car l’implémentation proposée semble répondre au besoin. En effet, les tests unitaires et les scénarios imaginés pour tenter de mettre en défaut l’algorithme ne semble pas de créer d’état incohérent la majorité du temps. Cependant, il est important de noter que des scénarios amenant à un état incohérent existent mais semble beaucoup trop improbable pour mériter d’être réellement considérer.

Pour citer quelques difficultés rencontrées. TODO

# Bibliographie

Documents étudiés lors de la phase de documentation : <https://github.com/MatthieuNICOLAS/2020-stage-membership/>

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

Site internet du LORIA : <https://www.loria.fr/fr/>

# 

# Annexe 1 : Capture d’écran de l’interface du prototype

Insérer capture

# 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 à été supprimé. |