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

Année universitaire 2019-2020

Remerciements

Table des matières

[Introduction 6](#_Toc42679057)

[Contexte 7](#_Toc42679058)

[MUTE 7](#_Toc42679059)

[Problème 7](#_Toc42679060)

[Etat de l’art 9](#_Toc42679061)

[Pair à pair 9](#_Toc42679062)

[Systèmes distribués 9](#_Toc42679063)

[Conflits et CRDT 9](#_Toc42679064)

[Protocole de membership 9](#_Toc42679065)

[Le protocole SWIM 11](#_Toc42679066)

[Prototype 13](#_Toc42679067)

[Première version 13](#_Toc42679068)

[Deuxième version 15](#_Toc42679069)

[Intégration dans MUTE 16](#_Toc42679070)

[Conclusion 17](#_Toc42679071)

[Bibliographie 18](#_Toc42679072)

[Annexe 1 : Capture d’écran de l’interface du prototype 19](#_Toc42679073)

[Annexe 2 : Messages 20](#_Toc42679074)

# Introduction

Présentation LORIA et COAST

Explication TRES vague du sujet

Annonce du plan

# 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 2

Client 1

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 :

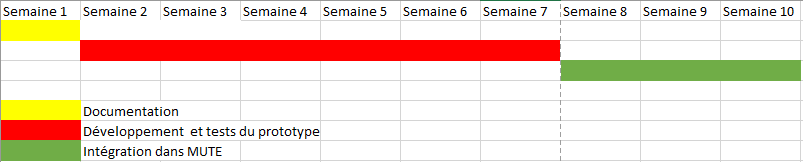
1. Un client ok peut très bien être perçu par un autre comme ko
2. Un client qui ne répond à un message n’est pas forcément ko

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 justement un protocole qui va permettre de suivre l’état des différents clients qui font partie du réseau (quel qu’il soit). 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.



# 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)

Nous allons donc dans un premier temps évoquer les bases à connaitre pour rentrer dans le sujet.

## Pair à pair

## 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 replciated 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, 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

Intro… Cf intro doc SWIM

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 vient par être déclarée défaillante et est exclue du réseau.

Pour faire parvenir à toutes les machines les informations relatives au statut 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 à connu 2 versions principales qui fournissaient 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) 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 peut de message différent, 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 pign-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 doit faire environ 2 secondes) et qui réalisera globalement le schéma détaillé dans la partie sur SWIM.

## Deuxième version

Plutôt axé bonne conception

# Intégration dans MUTE

?

# Conclusion

Objectifs atteint au moment de rendre le rapport et pronostic sur la fin du stage

Difficultés rencontrées

# Bibliographie

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

Insérer capture

# Annexe 2 : Messages

|  |  |
| --- | --- |
| Ping |  |
| Ping-req |  |
| Ack |  |
| Data-request |  |
| Data-update |  |
| Ping-reqRep |  |

|  |  |
| --- | --- |
| Joined |  |
| Alive |  |
| Suspect |  |
| Confirm |  |