LOG 735

Projet de session

Synchronisation dans les systèmes distribués

Par Pierre-Luc Delisle et Jérémie Beaulieu

Introduction

Un système distribué est un système composé de plusieurs noeuds qui communiquent entre eux sur un réseau. Ces noeuds, bien qu'indépendants, ont parfois la nécessité de se synchroniser entre eux afin d'accéder à une ressource particulière, que ce soit une base de données ou un dispositif matériel quelconque. Ainsi, il est important de développer des algorithmes de synchronisation robuste afin de synchroniser l'accès des noeuds à la ressource. Dans ce dernier laboratoire du cours LOG735, il nous a été demandé de concevoir une application de simulation permettant de mesurer les performances entre deux algorithmes :

- L'algorithme *Token Ring;*
- L'algorithme de Ricart-Agrawala.

Ce rapport fait état du développement de notre application. Nous présenterons d'abord le contexte de l'application. Nous décrirons ensuite sa conception et de son implémentation, pour finalement terminer avec la validation et une analyse des résultats obtenus.

Proposition et contexte du projet

Le contexte de notre application est très simple. L'application comporte plusieurs noeuds et un seul *device*. Ce *device* représente un calculateur ; seulement cet objet est en mesure et d'exécuter un certain calcul. Il pourrait s'agir d'un noeud équipé d'un GPU auquel on envoie des données par exemple, ou être équipé d'une plus grande puissance de calcul. Bref, le *device* n'est qu'un objet pouvant être attribué à un seul et unique noeud à la fois. Il s'agit donc de l'implémentation de notre section critique.

Nous proposons donc une application permettant de simuler deux algorithmes qui permettent d'assurer la sûreté de la section critique et ainsi de mesurer la performance entre les deux algorithmes.

Spécifications détaillés du projet

Pré-requis de fonctionnement	Java 1.8
Système d'exploitation	N'importe quel système d'exploitation supportant la version 1.8 de la SDK Java (Windows, macOS et Linux)
Compatibilité	Tous les fureteurs web modernes et à jour seront compatibles avec la version courante.
Logiciels requis	Apache Maven 3.5.0 pour installer les dépendances du projet.
Cadriciel utilisé	Java Spring Boot 1.5.4

Analyse et conception

Analyse des cas d'utilisation

Notre application ne comporte qu'un seul cas d'utilisation :

CU-01 : Démarrage d'une simulation

Valeur	Description
Portée	Application de simulation
Niveau	But de l'utilisateur
Acteur principal	Utilisateur
Parties prenantes et intérêts	Utilisateur : désire démarrer une simulation
Préconditions	Le serveur est démarré. L'utilisateur est sur la page principale de l'application
Postconditions	L'utilisateur peut consulter le résultat de la simulation.

Scénario principal :

Actions de l'acteur	Responsabilité du système
1. L'utilisateur clique sur le bouton de création d'une simulation.	
	2. L'application propose de rentrer le nombre d'hôtes dans le système.
3. L'utilisateur rentre le nombre d'hôtes, la latence désirée et la durée de la simulation.	
	4. L'application propose de lancer la simulation.
	5. L'application simule la synchronisation à

	l'aide des deux algorithmes simultanément.
6. L'utilisateur consulte les résultats.	

Caractéristiques de l'architecture et décisions de conception

L'architecture que nous avons opté pour cette application est peu conventionnel pour ce type d'algorithme. Nous avons opté pour une simulation dite sans état (*stateless*) sous forme d'API Web *RESTful*. En effet, l'équipe a tenté de repousser les technologies vues en classe durant la session au profit des nouvelles méthodes utilisées dans l'industrie. Ainsi, la synchronisation se fait entièrement via l'appel de méthodes REST (*endpoints*), sans l'utilisation de *Socket* ou autre alternatives de communication Java. La solution utilise un serveur Web *Java Spring Boot 1.5.4* au niveau du *back-end*. Cela nous permet de faire des appels à l'API via un *front-end* Javascript ou directement avec *Postman* afin d'interagir avec le système. Cette architecture est peu commune puisque, normalement, la synchronisation intervient principalement dans des systèmes dits *avec état* (*stateful*). Une architecture basée sur Java RMI aurait peut-être été une solution plus appropriée, mais le défi de vouloir repousser les notions vues en classe à un autre niveau avec une technologie beaucoup plus moderne a conclu le choix de celle-ci. De plus, cette voie était également beaucoup plus simple afin d'implémenter une page Web de visualisation de la simulation, ce qui était primordial pour les deux membres de l'équipe. Le Web étant désormais omniprésent de nos jours dans toutes les sphères du génie logiciel, il nous était impensable de faire autre chose qu'une application Web pour ce projet, bien que cela dépassait certainement les attentes et multipliait la complexité d'implémentation. De plus, les membres de l'équipe étaient largement plus à l'aise avec des technologies Web *RESTful* que les technologies vieillissantes présentées dans le cadre du cours.

Ainsi, notre solution est composée principalement d'un API REST permettant d'interagir directement avec l'état des composants de la simulation. Par exemple, plusieurs méthodes REST ont été implémentées afin de modifier l'état d'un noeud afin que celui-ci, par exemple, passe d'un état passif à une demande d'accès à la section critique.

La solution, puisque Web, est divisée en plusieurs paquetages :

configurations	Classes regroupant les diverses configurations de notre API et de la documentation.
factories	Objets permettant d'instancier des POJOs.
pojos	Plain Old Java Objects, des objets représentant des entitées Java.
exceptions	Exceptions personnalisées pouvant être levées dans notre application.
rest.api.v1	Paquetage regroupant toutes les classes reliées à la portion Web de l'application
responses	Classes représentant des réponses HTTP pouvant être automatiquement *marshal* en JSON avec le cadriciel Java Spring.
resources	Les classes représentant les méthodes REST de l'application pouvant être accessibles à l'extérieur.
services	Paquetage regroupant les fils d'exécution (*thread*) de simulation ainsi que les services internes de l'application.
utils	Classes *runnables* ou utilitaires.

En regroupant ces classes ainsi, il est beaucoup plus facile de se retrouver dans le projet. Chaque paquetage a un rôle précis représentant la nature intrinsèque de la classe.

Diagrammes de classes et de séquences

Afin d'améliorer la lisibilité de ces diagrammes, voir les fichiers joints lors de la remise du rapport ou directement dans le code source de l'application dans le dossier *diagrams*.

Implémentation

Algorithmes

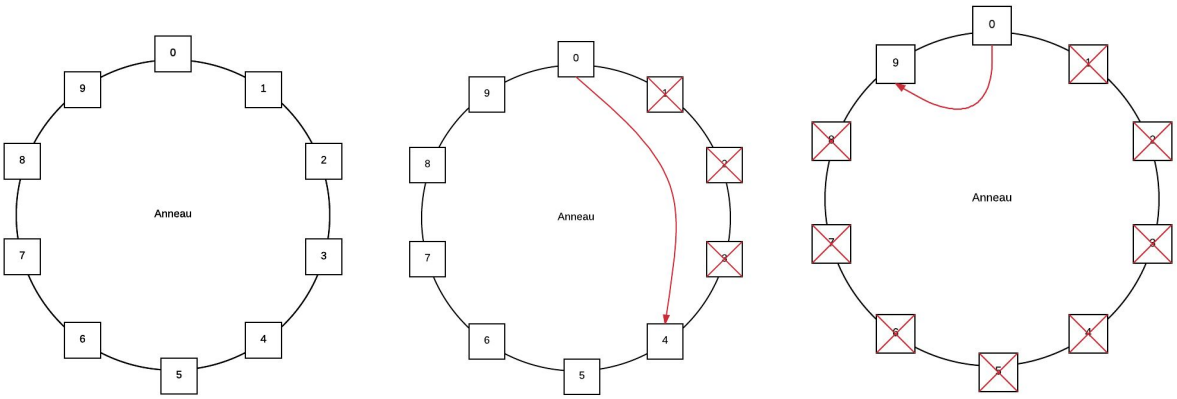
Token Ring

L'algorithme de synchronisation de l'anneau et du jeton, ou plus communément appelé *Token Ring Synchronization for distributed systems,* fut le premier algorithme implémenté par l'équipe. Pour ce faire, l'équipe a dû bâtir de toute pièce l'application en suivant la conception et l'élaboration est paquetages décrits précédemment. Ainsi, une classe *Node* fut d'abord créée, celle-ci représentant un Noeud. Il s'agit de cet objet qui sera instancié de multiples fois lors de chaque simulation. Cette classe de type POJO comporte tous les attributs nécessaires pour connaître l'état du noeud, à savoir son numéro d'identification, s'il est en possession ou non du jeton (*token*), s'il demande un accès à la section critique (*requestingCriticalSection*), s'il est en activité ou non (*down*), et son état global. Cet objet comporte également une file d'attente (*queue*) de travail (*Job*). Le POJO *Job* est un objet simulant un calcul devant être effectué. Il s'agit de deux chiffres aléatoires devant être additionnés. Comme mentionné dans la présentation du contexte de l'application, seul l'objet *Device* peut exécuter le calcul.

Une classe *Message* est l'objet qui est transigé entre les noeuds. La structure du message est comme suit :

Champ	Description
sourceID	L'identifiant unique du noeud source issuant le message.
destinationID	L'identifiant unique du noeud de destination recevant le message.
message	Une *String* en texte clair.
encryptedMessage	Un texte crypté à l'aide de l'algorithme symétrique AES-256.
timestamp	Utilisé seulement dans le cadre de l'algorithme Ricart-Agrawala

L'algorithme débute donc avec l'attribution du jeton au noeud portant l'identifiant 0 ou, si ce dernier est inactif, au premier noeud actif en commençant par le numéro d'identification le plus bas. Le premier noeud crée un premier message crypté comportant le message «*Passing token*». Ce message est envoyé au noeud portant le numéro d'identification supérieur à lui-même de 1 en passant par la ressource du noeud situé dans la classe *NodeResource* de l'API REST. Cette méthode appelle le *RingService* où toutes les méthodes relatives à l'anneau se retrouvent et traite le message. S'il s'agit bel et bien d'un passage du jeton, l'attribut déclarant que le noeud a en sa possession le jeton est mis à *vrai*. Si le noeud possédant désormais le jeton a demandé l'accès à la section critique, celle-ci lui est accordée. Il peut donc utiliser le singleton *Device* afin d'effectuer le premier calcul de sa file d'attente FIFO. La réponse est affichée en console, puis le jeton est passé au suivant en répétant la même séquence. Un fait à noter est qu'en raison du design REST et sans état, n'importe quel noeud peut parler à n'importe quel autre en cas de panne. En effet, si nous avions utilisé le principe de Socket, chaque noeud aurait eu son propre fil d'exécution en connaissant explicitement les coordonnées du noeud suivant en ayant un Socket de communication ouvert avec celui-ci ou, au mieux, avec le second noeud suivant en cas de non-réponse du noeud suivant. Dans notre cas, le noeud suivant est déterminé par le *thread* de simulation qui teste si le prochain noeud est actif. Le prochain noeud actif est ainsi trouvé, qu'il s'agisse du prochain noeud de l'anneau ou celui juste à la gauche du noeud ayant actuellement le jeton.



Le prochain noeud actif trouvé est donc envoyé à la méthode *passToken* afin de passer le jeton à ce dernier. Aucun noeud ne tient en mémoire les coordonnées du prochain noeud, ce qui permet d'être sans état et totalement indépendant de l'algorithme dans lequel il évolue, comme il se doit normalement dans un système distribué moderne.

Une seule *Job* n'est exécutée à la fois dans la section critique. Cela permet d'assurer le principe de la vivacité puisqu'un noeud n'attendra pas indéfiniment en quête de la section critique, celle-ci se libérera après qu'un seul calcul n'ait été effectué sur le *Device*.

Puisque REST ne supporte pas nativement les communications suivant un anneau, l'entièreté de l'algorithme est programmée dans un fil d'exécution de simulation dans la classe *TokenRingSimulationThread*.

Ricart-Agrawala

L'algorithme de Ricart-Agrawala reprend l'architecture décrite précédemment et vient simplement ajouter de nouvelles méthodes REST à l'API. Toutefois, le manque de temps a commencé à se faire sentir chez les membres de l'équipe et l'implémentation ne suit pas nécessairement les meilleures pratiques. Toutefois, l'algorithme est fonctionnel et tolérant aux fautes. Tout comme Token Ring, l'algorithme de Ricart-Agrawala permet de synchroniser le Device pour que celui-ci ne soit acquis que par un et un seul noeud à la fois. Le fil d'exécution de la simulation, présent dans la classe *RicartAgrawalaSimulationThread*, permet de vérifier de façon continuelle si un noeud demande la section critique. Lorsqu'un noeud demande un accès à celle-ci, un message comportant la mention cryptée *«Request»* est créée et envoyé à tous les autres noeuds faisant partie de la simulation. Chacun des noeuds répond par le message crypté *«Reply»* en fonction des conditions de l'algorithme. Si toutes les réponses reçues sont effectivement des mentions *«Reply»*, l'accès à la section critique est accordé et le noeud est libre d'exécuter son calcul. La tolérance de faute est assurée, car le thread de simulation s'assure, avant d'envoyer le message, que l'hôte destinataire est actif. En cas contraire, l'algorithme ignore ce noeud et le noeud demandant la section critique n'attendra pas indéfiniment une réponse. Toutefois, nous n'avons pas de mécanisme permettant de repérer automatiquement un noeud en problème, plus communément appelé time out. Il s'agit un des points faibles de notre implémentation.

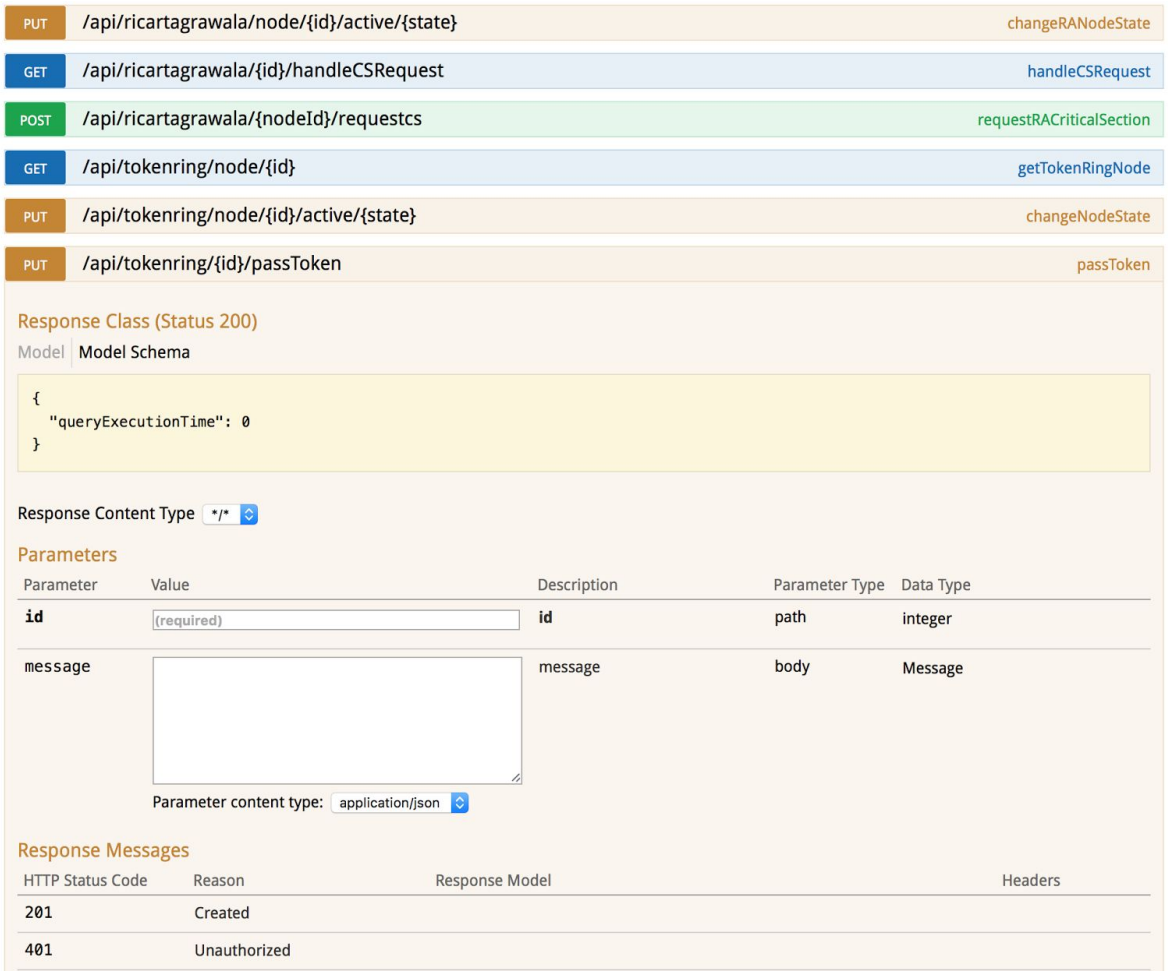
Configuration requise

Aucune configuration n'est requise pour faire tourner cet algorithme, de même que *Token Ring*.

Critères de qualité

Documentation

Un des critères majeurs d'un bon API est sa documentation. Pour ce faire, l'outil Swagger a été utilisé afin de documenter automatiquement les méthodes REST de l'API. Ainsi, cela donne une documentation complète et détaillée des appels des méthodes.



Cette documentation est accessible à l'URL http://localhost:8080/swagger-ui.html lorsque le serveur *Spring* est en marche.

Extensibilité en fonction du contexte du projet

Au niveau de ce critère, l'extensibilité se traduit dans notre projet par la création du nombre de noeuds que l'utilisateur désire avoir dans la simulation. Ce nombre n'est pas limité et donc permet une certaine extensibilité.

Tolérance aux fautes

Dans le cas de l'algorithme *Token Ring*, une tolérance aux fautes a été introduite comme mentionné précédemment. Cette tolérance aux fautes permet de faire passer le jeton d'un noeud à un autre en sautant autant de noeuds inactifs qu'il y en a dans la simulation, mais exige toutefois un minimum de deux noeuds actifs. Dans le cas de l'algorithme de Ricart-Agrawala, l'algorithme vérifie avant d'envoyer le message si le noeud est actif. S'il ne l'est pas, aucun message ne sera envoyé et donc la suite de l'exécution n'attendra pas le message de ce noeud. Un noeud peut donc tomber inactif sans que cela n'influence le déroulement de l'exécution.

Transparence

L'application n'étant pas une application distribuée à proprement parler, le principe de transparence n'avait pas à être pris en compte. Cependant, puisque l'utilisateur utilise un fureteur Internet, celui-ci a l'impression de dialoguer avec un serveur standard, comme n'importe quelle application client/serveur.

Performance

Encore une fois, il ne s'agit pas réellement d'une application distribuée. Cet aspect des systèmes distribué n'a pas eu à être pris en compte lors de la conception.

Validation et vérification

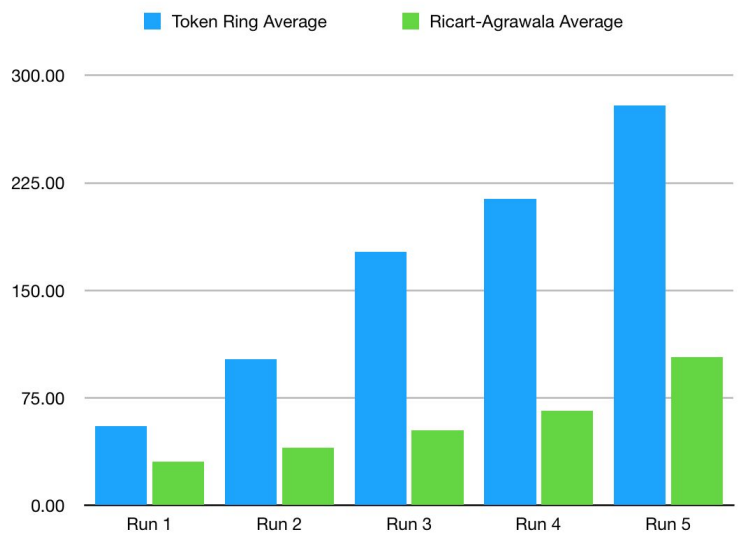
Plan de tests

Les tests ont principalement été effectués à la main avec l'outil *Postman* ou en «*debug*». Il s'agit probablement du plus gros point faible de la solution présentée. Par manque de temps, aucun test n'a été codé.

Toutefois, nous avons été en mesure de comptabiliser des résultats de tests de performance. Nous avons comparé le temps nécessaire moyen, à latence et nombre de noeuds égal, pour qu'une tâche puisse être exécutée.

Résultats et analyse des résultats

Afin d'analyser correctement les deux algorithmes, nous avons effectué plusieurs tests qui font varier chacun un paramètre ; la latence ou le nombre de noeuds. Voici les résultats pour la variation du nombre de noeuds :



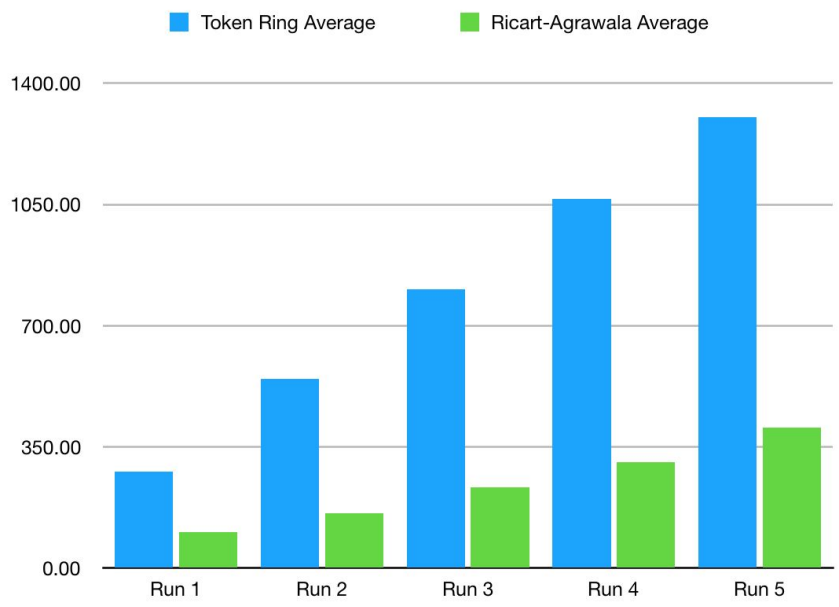
Variation of number of nodes

	Token Ring Average	Ricart-Agrawala Average
Run 1	55.39	30.33
Run 2	102.15	40.43
Run 3	176.92	52.22
Run 4	213.73	66.01
Run 5	278.76	103.48

Run table

	Number of nodes	Latency	Duration
Run 1	10	10	60
Run 2	20	10	60
Run 3	30	10	60
Run 4	40	10	60
Run 5	50	10	60

Voici les résultats pour la variation de la latence :



Variation of latency of nodes

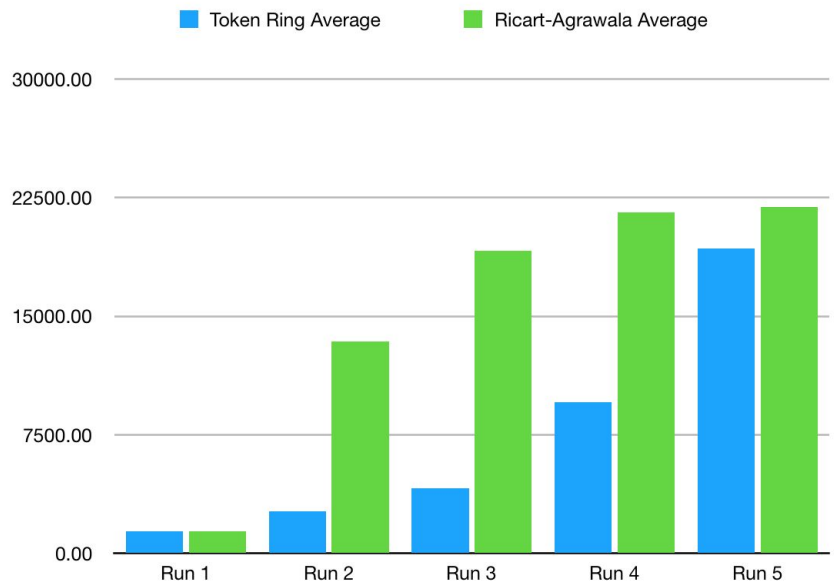
	Token Ring Average	Ricart-Agrawala Average
Run 1	278.76	103.48
Run 2	546.81	158.67
Run 3	806.14	232.76
Run 4	1066.71	307.04
Run 5	1301.94	406.08

Run table

	Number of nodes	Latency	Duration
Run 1	50	10	60
Run 2	50	20	60
Run 3	50	30	60
Run 4	50	40	60
Run 5	50	50	60

Comme on peut le constater, l'algorithme *Token Ring* est beaucoup plus lent que Ricart-Agrawala dans les deux cas de tests. Cela est dû au fait que le jeton doit se rendre jusqu'au noeud désirant avoir la section critique, alors que le jeton peut être très loin de ce noeud en question lorsque celui-ci demande l'accès à la section critique. Pour l'algorithme de Ricart-Agrawala, il est plus rapide d'envoyer un message à 50 noeuds et d'analyser les réponses par la suite que d'attendre son tour comme dans l'algorithme *Token Ring*.

D'autres tests ont par la suite été réalisés :



Random runs

	Token Ring Average	Ricart-Agrawala Average
Run 1	1417.25	1377.68
Run 2	2635.54	13376.29
Run 3	4134.48	19126.80
Run 4	9554.82	21546.45
Run 5	19260.17	21902.96

Run table

	Number of nodes	Latency	Duration
Run 1	500	5	60
Run 2	750	5	60
Run 3	1000	5	60
Run 4	1000	10	60
Run 5	1000	20	60

On peut voir que lorsque le nombre de noeuds explose, c'est *Token Ring* qui remporte. Il est dorénavant plus long pour Ricart-Agrawala d'envoyer des messages à autant de noeuds et d'analyser ensuite les réponses. Toutefois, l'écart s'amenuise lorsqu'on augmente la

latence. *Token Ring* semble donc plus sensible aux variations de latences que Ricart-Agrawala.

Analyse des forces et faiblesses de la solution

Une force de notre solution serait son implémentation RESTful. Au lieu d'utiliser des technologies pour le moins désuètes telles que Java RMI, voir même Corba, nous avons opté pour la modernité et la simplicité d'une implémentation Web RESTful. Cela nous permet également d'avoir un visuel intéressant sur une page Web et d'interagir manuellement avec les algorithmes afin de leur soumettre des conditions différentes. Une architecture REST permet de moduler soit même une simulation pour voir le comportement en temps réel des algorithmes.

Cette force est également une faiblesse au niveau de la performance. REST s'appuie sur une connexion TCP/IP. Ainsi, à chaque appel du *front-end*, cela implique une certaine latence avec le serveur. Toutefois, cela est négligeable puisque, lorsque la simulation est en cours, les méthodes sont appelées directement du fil d'exécution de la simulation (*RicartAgrawalaSimulationThread* ou *TokenRingSImulationThread*) au *endpoint*, sans passer par le serveur Web Spring.

Une autre faiblesse serait que nous n'avons aucun test automatisé afin de tester des cas limites de notre solution, soit avec un nombre faramineux de noeuds ou encore avec des requêtes en rafale sollicitant la mémoire. Ainsi, il nous a été difficile de tester si la demande en mémoire de l'application est relativement correcte. Si les requêtes s'accumulent plus rapidement que ce que la simulation est capable de traiter, il se peut qu'il y ait une erreur du type *Stack Overflow* ou carrément un manque de mémoire de la machine virtuelle Java. Toutefois, ce risque est relativement minime dans le cadre d'une simulation normale. Nous aurions pu toutefois implémenter des mécanismes permettant de réguler les demandes d'accès à la section critique.

Aussi, la solution, bien que compartimentée en paquetages, comporte beaucoup de couplage. L'utilisation d'interfaces au niveau des simulations aurait certainement diminué

le couplage entre les classes. Le processus de conception aurait dû être revu plus tôt dans le développement de l'application.

De plus, notre conception autour des principes REST vient un peu coller le principe *stateless* dans des algorithmes qui se veulent intrinsèquement du type *stateful*. Il s'agit donc d'une conception un peu spéciale pour ce type d'usage. Nous tenions toutefois à utiliser des technologies modernes pour réaliser ce laboratoire afin d'augmenter légèrement l'intérêt de la part des membres de l'équipe à réaliser ce laboratoire qui, comme nous le verrons plus tard dans la conclusion de ce rapport, est pratiquement inutile.

Finalement, la simulation aurait pu avoir un mode entièrement manuel ou avoir une option afin de basculer entre un mode automatique et manuel. Toutefois, l'équipe s'est fait plutôt diriger dans une voie où l'entièreté de la simulation devait se faire automatiquement.

Recommandations d'amélioration futures

Comme mentionné précédemment, un réusinage de la solution actuelle au niveau de sa conception pourrait être envisagé dans une itération future, de même que l'ajout d'un mode manuel qui fournit des résultats de façon dynamique. Toutefois, l'équipe a manqué de temps afin d'implémenter ces fonctionnalités. L'équipe a pris une semaine de retard au début du laboratoire afin de réfléchir à la solution et de partir un code fonctionnel. Si l'équipe avait disposé de ce temps supplémentaire, ces éléments auraient pu être ajoutés à la solution présentée. La création de tests automatisés fait également partie des améliorations dont l'application bénéficierait dans une itération ultérieure. Le développement d'un outil intégré à la solution permettant de surveiller l'utilisation de la mémoire serait également un bon ajout.

De plus, nous aurions pu faire tracer les graphiques de performance directement dans la page Web à la suite de l'exécution de la simulation. Nous aurions pu utiliser la librairies d3.js pour tracer les graphiques. Cet ajout pourrait se faire dans une itération future.

Conclusion

Nous avons présenté dans ce rapport une solution permettant de comparer deux algorithmes de synchronisation dans les systèmes distribués, soit l'algorithme Token Ring et de Ricart-Agrawala. Cette solution se veut RESTful, stateless et orientée Web, trois principes fondamentaux d'un système distribué actuel qui ne sont malheureusement pas abordés dans le cadre du cours. Toutefois, l'équipe fut particulièrement déçue de la proposition de ce laboratoire qui, admettons-le, est franchement inutile de nos jours. Il aurait été largement plus profitable non seulement pour le développement des connaissances des membres de l'équipe, mais également pour l'application des concepts d'un système distribué, de développer une application Web RESTful distribuée avec des cadriciels tels que Apache Spark, Mesos ou Cassandra dans le cas d'une base de données distribuée. De nombreux autres cadriciels peuvent également être utilisés pour développer une application distribuée orientée Web. Toutefois, ce laboratoire ne proposait pas un contexte permettant l'utilisation de ces cadriciels et le cours ne mentionnait même pas l'existence de ces cadriciels très intéressants pour la réalisation de systèmes distribués, démontrant par le fait même l'obsolescence de celui-ci. L'application développée est pratiquement inutile dans le monde réel et peu intéressant non seulement dans le cadre d'un laboratoire portant sur les systèmes distribués, mais également dans une perspective professionnelle. Rappelons qu'à la base, cette application n'est même pas un système distribué, les noeuds étant de simples instances en mémoire et l'application tournant sur une seule machine physique. Certes, en fin de compte, cette nuance a très peu d'impact sur les concepts orientés système distribués qu'embarque l'application. Pour être en présence d'un véritable système distribué, il nous aurait fallu développer une architecture de type microservice représentant un seul noeud puis, avec un orchestrateur tel que Docker Swarm, multiplier et administrer ces noeuds et les faire communiquer entre eux. La complexité du système aurait toutefois augmenté de façon exponentielle, sans même savoir si l'implémentation des algorithmes serait réalisable dans un tel contexte stateless. De plus, la contrainte de temps de quatre semaines afin de réaliser ce tel système aurait été très l'imitatrice et aurait rendu la création de cette solution très difficile à faire, sans mentionner non plus que ce type de solution nous a été déconseillé par le professeur du cours, ce dernier voulant une solution plus simpliste comme celle présentée, mais qui, par

le fait même, n'est fondamentalement pas distribuée. De plus, l'équipe doute très fortement de revoir ces algorithmes dans la nature une fois sur le marché du travail, les architectures de systèmes distribués actuels favorisant grandement la nature stateless du Web moderne, une conception sous forme de microservices REST avec peu ou sans dépendances externes et facilement extensibles sur une infrastructure du type Amazon Web Services. Ces types de systèmes sont de véritables systèmes distribués modernes réellement utilisés dans l'industrie et développer une solution de ce type aurait été largement plus profitable que ce laboratoire. Le témoignage d'un autre professeur de l'école ainsi que la présentation vue en classe lors du dernier cours viennent confirmer nos dires. En résumé, ce laboratoire loupe complètement le but proposé et étudié dans le cours durant toute une session. Un laboratoire favorisant la modernité, le principe stateless et les types synchronisations qui en découlent, les technologies Web serait largement plus intéressant. S'éloigner du Web en 2017 est littéralement se cacher la tête dans le sable puisqu'il s'agit de l'intergiciel le plus utilisé au monde. En effet, HTTP et REST sont indépendants de technologie et comportent peu de problème d'encodage et de format.

L'équipe a toutefois apprécié la liberté technologique qu'offrait ce laboratoire.