Le génie pour l'industrie

Département de génie logiciel et des TI

Rapport de laboratoire

Cours	LOG735
Session	Eté 2015
Groupe	01
Laboratoire	LABORATOIRE 1 – Synchronisation et concurrence dans les systèmes distribués
Chargé de laboratoire	Thierry Blais
Professeur	Lévis Thériault
Étudiant(e)(s)	Moreau Max Charly Simon
Adresse(s) de courriel	max.moreau.1@ens.etsmtl.ca charly.simon.1@etsmtl.net
Code(s) permanent(s)	MORM30038905 SIMC28069108
Date de remise	14 mai 2015

1 Tables des Matières

2 Introduction

Ce laboratoire sert a nous familiarisé a la notion de socket, de client-serveur et met l'accent sur les différents problèmes sous-jacent a leur conceptions.
Nous créerons différents package pour chaque nouveaux retirement tant niveau serveur que client nous permettant par la suite de retracer aisément les changements engendre pour des petites demandes.
Nous aborderons également les matières vu en cour pour les systèmes distribue en particulier la transparence et la synchronisation.

3 Réponses aux questions

3.1 Question 1)

Expliquez pourquoi la seconde instance de Client ne reçoit pas la chaîne « BONJOUR» du Serveur, en fonction de la gestion du threading employée par la version initiale de l'application.

Il n'y a pas de réponse parce que le serveur n'est pas en mode multithread ou multiproces.
Son exécution actuelle est coincer dans le while du client1. (Tant que le client 1 n'a pas envoyer une fin de connections).
Lorsque le client1 envoie une fin de connections on écrit bye puis FERME le serveur.
Conclusion on n'aura jamais 2 client avec ce code.
Lorsque le client2 se connecte il est coincer dans le serverSocket.accept(), qui semble être java qui le met en attente jusqu'à ce que le client1 se libère.

3.2 Question 2)

Expliquez sommairement les modifications effectuées en indiquant quelles méthodes ont été modifiées, toujours en mettant l'emphase sur la gestion du threading dans votre version modifiée de l'application.

Une liste exhaustive des différences est disponible dans l'annexe.
Pour résumé nos différences nous avons ajouté les fonctionnalité suivante :
Serveur :

- Chaque requête est traité dans un thread différent, permettant ainsi de supporté une multitude de client 'simultanément'.
- Le serveur ne se ferme plus après avoir servit un client. Mais tourne de façon 'infinie' comme tous vrai serveur se doit.
- Nous utilisons l'idiome try-catch-finally afin de géré proprement notre mémoire des socket.
- Si un accept échoue on passe a la prochaine connection et ne coupe pas le serveur.
- Nous informons le client lorsque nous terminons la connection. (par un bye).
- Le serveur ne gere pas le cas ou nous atteignons le maximum de thread possible.

Client :

- Lorsque nous recevons le message de fin de connection nous fermons proprement le socket du client afin de ne pas écrire dans le vide. (et évité ainsi les erreurs inutiles).

NB : Nous avons envisager une version multiprocess qui aurait l'avantage d'être plus performante que le multi-threading, mais la question mentionnait de faire l'emphase sur le threading donc nous avons choisit celle-ci.

3.3 Question 3)

Expliquez les modifications effectuées tout en justifiant votre démarche par rapport à la notion de transparence d'accès des systèmes distribués. Indiquez où vous avez situé les coordonnées (adresse et port) de la seconde instance et justifiez cette décision. Analysez l'impact de cette décision en fonction des différents types de transparence applicables.

Nous avons instancié notre second serveur avec ces paramètres :
adresse : 127.0.0.2
port : 10118.

Nous avons choisit de mettre notre instance sur une autre adresse Ip car nous supposons que si le serveur1 ne répond plus est que la machine est saturé ou que le réseau pour cette route l'est.
Dans tous les cas ceci nous simule comme un host non joignable d'où le fait que nous avons choisit de définir le serveur a une autre adresse plutôt qu'un autre port.
Pour le port, nous avons choisit de définir le même de tel sorte que ceci soit reconnu tel un service. (par exemple si notre DNS est indisponible on va sur le prochain host et l'on demande encore au port 53).
Nous aurions néanmoins pu changer le port de notre second serveur mais il nous aurait fallu un service d'annuaire que le client consulterais pour savoir qui contacter. (ou un annuaire prédéfinit)
Ceci serait néanmoins la solution la plus adéquate pour représenté la dynamique des services sur un réseau.
Une liste exhaustive des différences est disponible dans l'annexe.

Diffèrent type de transparence / impact :

Type de transparence	**Description**	**Impact**	**Choix**
Accès	Permet aux ressources locaux et distantes d'être accédées en utilisant les mêmes opérations	Aucun, il n'y a aucun objet distant pour le moment.	Non utile
Localisation	Permet aux ressources d'être accédées sans connaître leurs localisations	Nécessite un service annuaire ou discovery.	Prepare. Le port commun peut servir d'un bon point de depart pour un discovery.
Concurrence	Permet à plusieurs tâches d'opérer en concurrence sans interférence	Aucun les taches sont independantes.	Aucun changement requis
Duplication	Permet à plusieurs instances du ressource d'être utilisées pour augmenter la fiabilité et la performance sans connaissance de la duplication de la part des utilisateurs et des applications	Spawn de plusieurs process sur l'host. Besoin d'un repartiteur.	Non le client est au courant.
Défaillance	Permet la dissimulation des défauts	Necessite des backup.	Implémentation du dictionnaire et TIMEOUT.
Mobilité	Permet le mouvement des ressources et des clients du système sans affecter le fonctionnement des programmes	Necessite une strategie de transfert. Tranfert progressif des sockets.	Non fait. (overkill par rapport au traitement)
Performance	Permet au système de se configurer pour améliorer la performance lorsque la charge varie	Necessite de l'apprentissage ou des strategies de contingence.	Non fait (overkill par rapport au traitement)
Extensibilité	Permet au système et aux applications de s'étendre sans changer la structure du système ou les algorithmes des applications	Aucun compte tenu du traitement des serveur.	Implementation du dictionnaire.

3.4 Question 4)

En vous basant sur votre version intermédiaire Q3, répondez à la question suivante : a-t-il été nécessaire de synchroniser les serveurs et pour quelles raisons?

Non, parce qu'il s'agit de 2 serveur echo simple.
En admettant néanmoins qu'il s'agisse de job plus important, lorsque notre timeout intervient au niveau du client et que l'on switch du serveur1 au serveur2. On pourrait notifier du serveur2 au serveur1 qu'on s'occupe a présent du job et de l'interrompre ou le discarder si il était toujours dans sa jobqueue.
Ceci serait uniquement dans le but d'une petite optimisation.
Dans tous les cas nous n'avons pas besoin que les serveurs commencent leur job au même tick de temps (donc pas de synchro de temps).

3.5 Question 5)

Expliquez sommairement les modifications effectuées en indiquant quelles méthodes ont été modifiées.

On a rajouter un 'static nb_req' dans le serveur afin de retenir le nombre de requête servit.
Une liste exhaustive des différences est disponible dans l'annexe.

3.6 Question 6)

Expliquez sommairement les modifications effectuées en indiquant quelles méthodes ont été modifiées.

Une liste exhaustive des différences est disponible dans l'annexe.
Avant chaque traitement, les serveurs communique entre eux afin de savoir le nombre de requêtes effectuées.
Les justifications de cette méthode sont dans la Q7 et Q8

3.7 Question 7)

Expliquez la différence majeure entre la version intermédiaire de la question 3 et la version de la question 6 en fonction de la notion de synchronisation. Détaillez les mécanismes de synchronisation implémentés à la question 6.

Les diagrammes d'interactions permettent de mieux jugé la différence entres ces 2 questions. Ceux-ci sont visibles dans la partie diagrammes.
(cf 'Diagramme Q3 et Q6')

Dans la Q6 nous synchronisons notre numéro de requête avec les différents serveur afin que ceux-ci transmette toujours le bon numéro.
Nous synchronisons avant le traitement car si notre serveur tombe en panne pendant le traitement nous voulons que la prochaine requête ai le numéro suivant. Ce qui signifie que nous avons du incrémenté et propager des la réception d'une requête.

3.8 Question 8)

Écrivez une discussion concise qui porte sur les notions abordées dans le laboratoire ainsi que sur l'extensibilité de la version de la question 6, dans l'éventualité où nous voudrions ajouter des serveurs de relève supplémentaires.

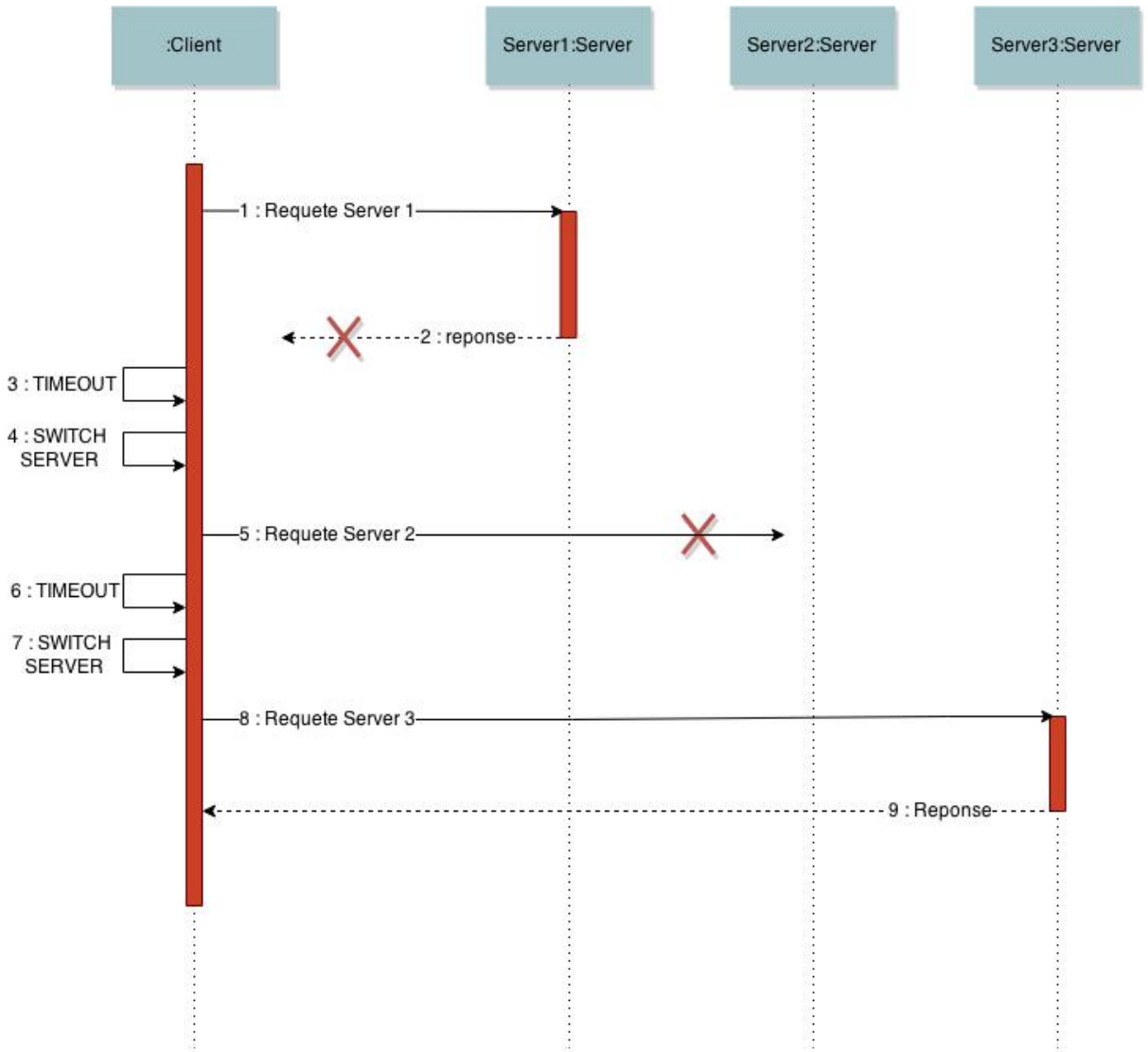
Plus l'on a de nouveau serveur plus les messages de synchro deviennent grand par rapport au traitement.
En admettant que l'on est n serveurs on a 'n' message de synchro pour 1 requêtes. O(1) vs O(n)
(cf 'Diagramme synchro naïve'')

Néanmoins on pourrais distribuer cette synchronisation ou l'externaliser.
Un peu tel un arbre, notre serveur informe 2 autres serveur de son nouveaux numéro, qui eux informe 2 autres ainsi de suites.
On peut aussi juste choisir que le serveur informe un autre du nouveaux numéro et lui demande d'effectuer la propagation a sa place. (différentes méthodes sont possible).
(cf 'Diagramme arbre de synchro' et 'Diagramme arbre agent').

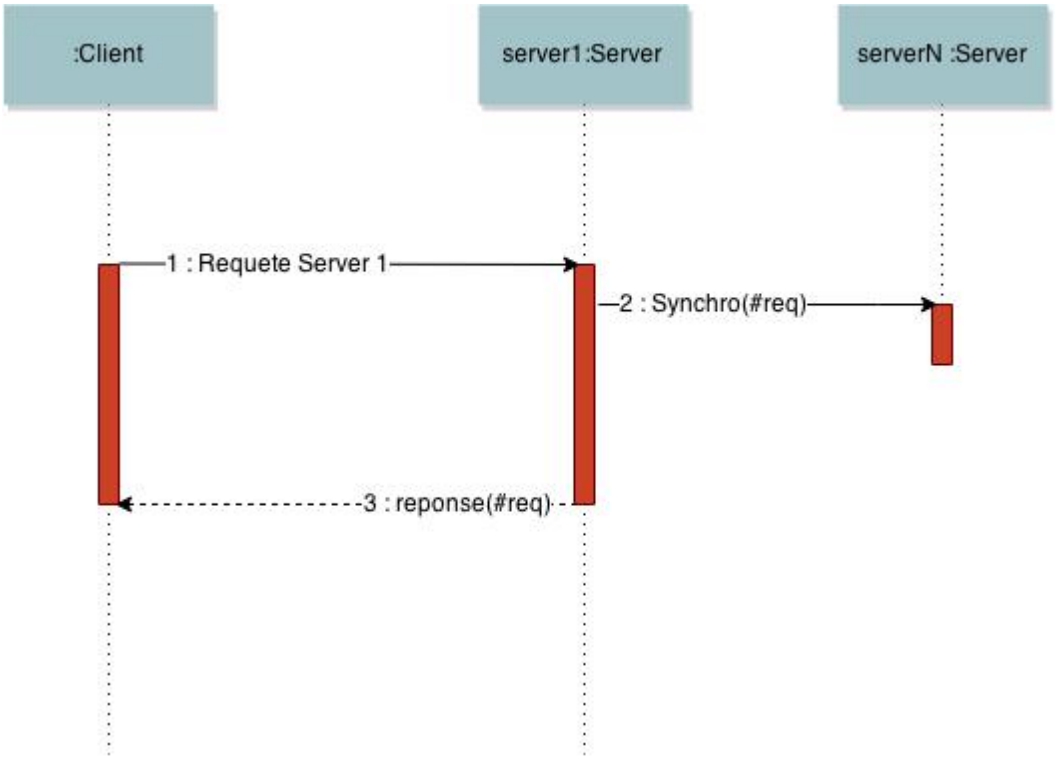
Afin d'accroître notre transparence pour le client nous pourrions également ajouter un serveur annuaire qui s'occuperait de référence la liste des serveurs actif sur lequel le client pourrait switcher mais ceci reporterais l'aspect critique de la disponibilité sur ce serveur Annuaire.
Une autre option serait de définir un protole comme par exemple ARP qui propagerais un message pour recherche les serveurs actifs et les ajouterais dans son dictionnaire.
Ceci a l'avantage de ne pas être dépendant de quelconque serveur actif mais demande plus de message réseaux.

4 Diagrammes

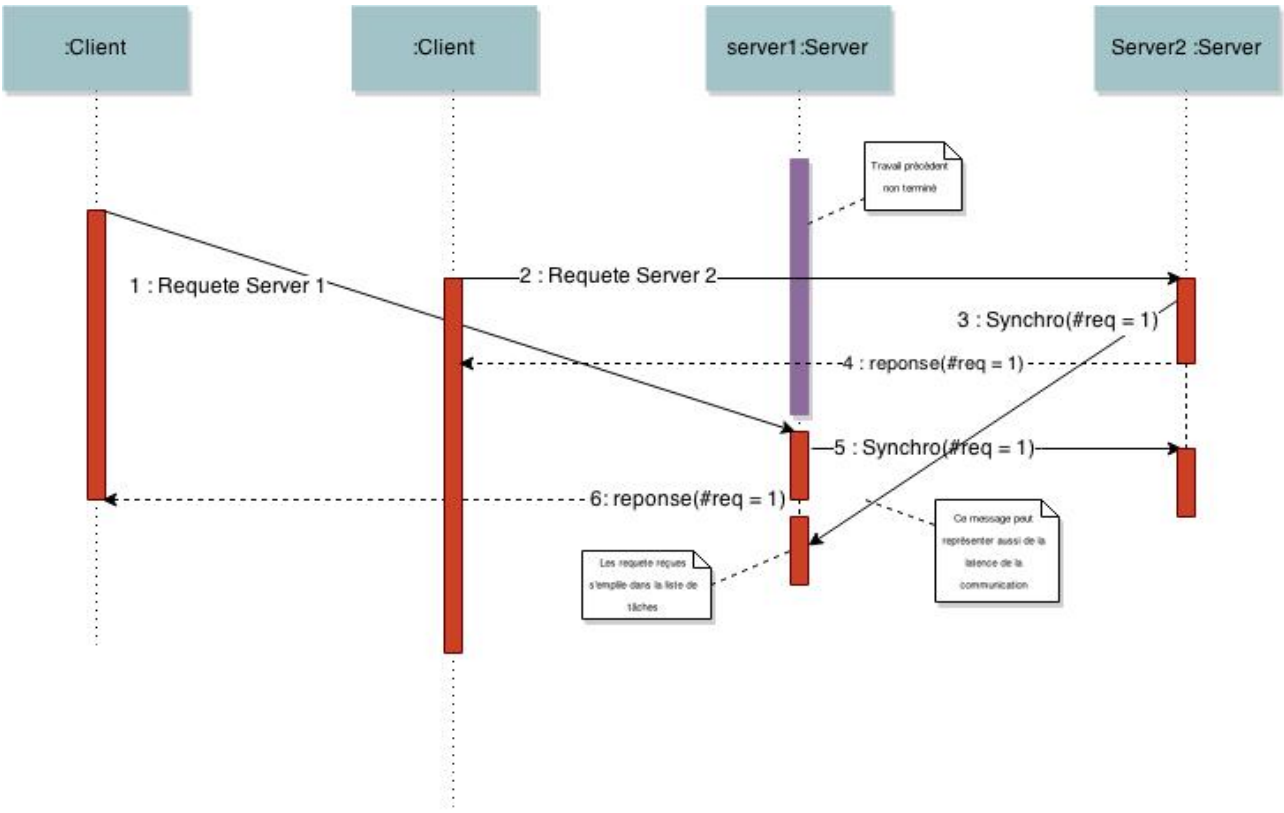
4.1 Diagramme Q3 :



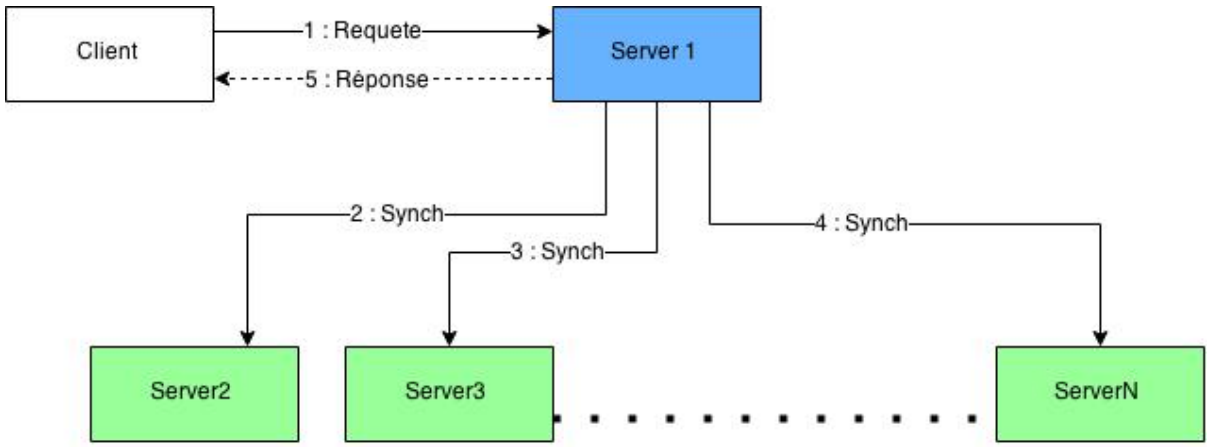
4.2 Diagramme Q6



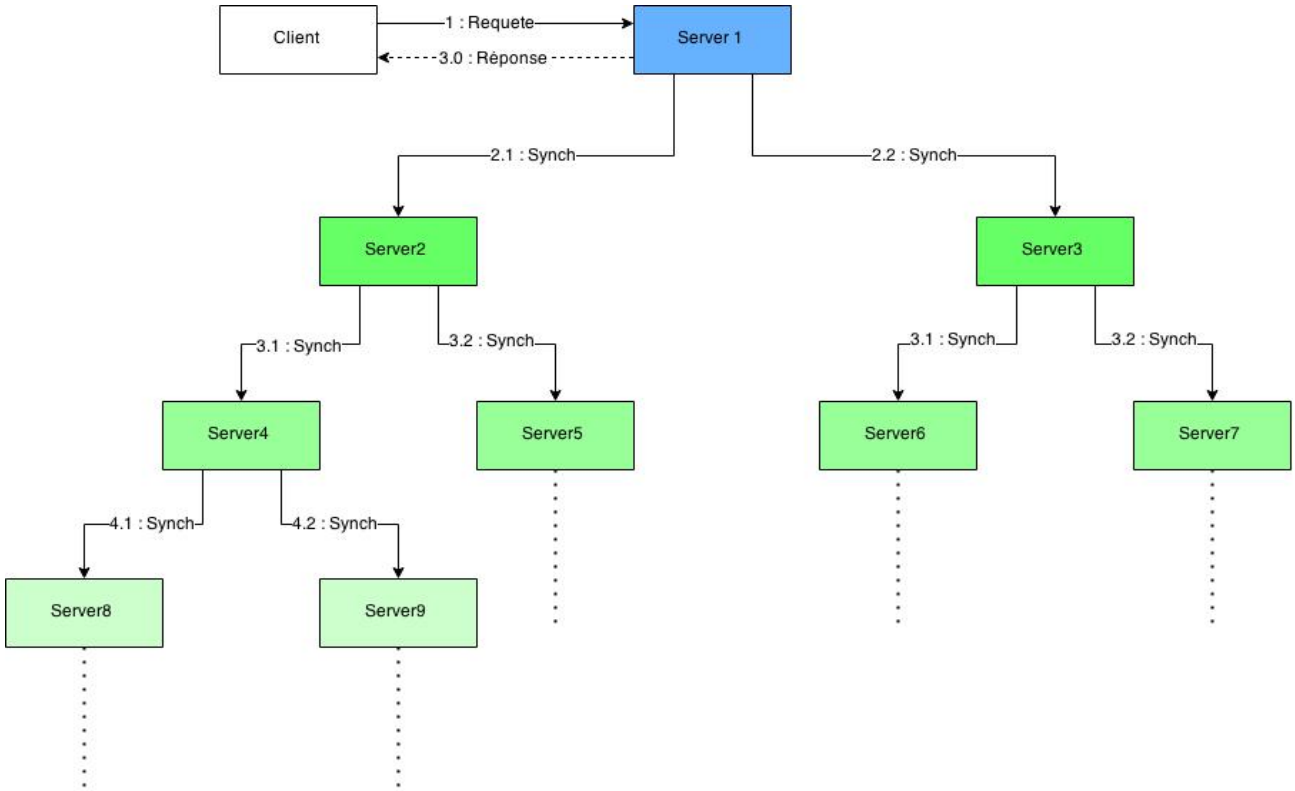
4.3 Diagramme : Erreur probable de synchro



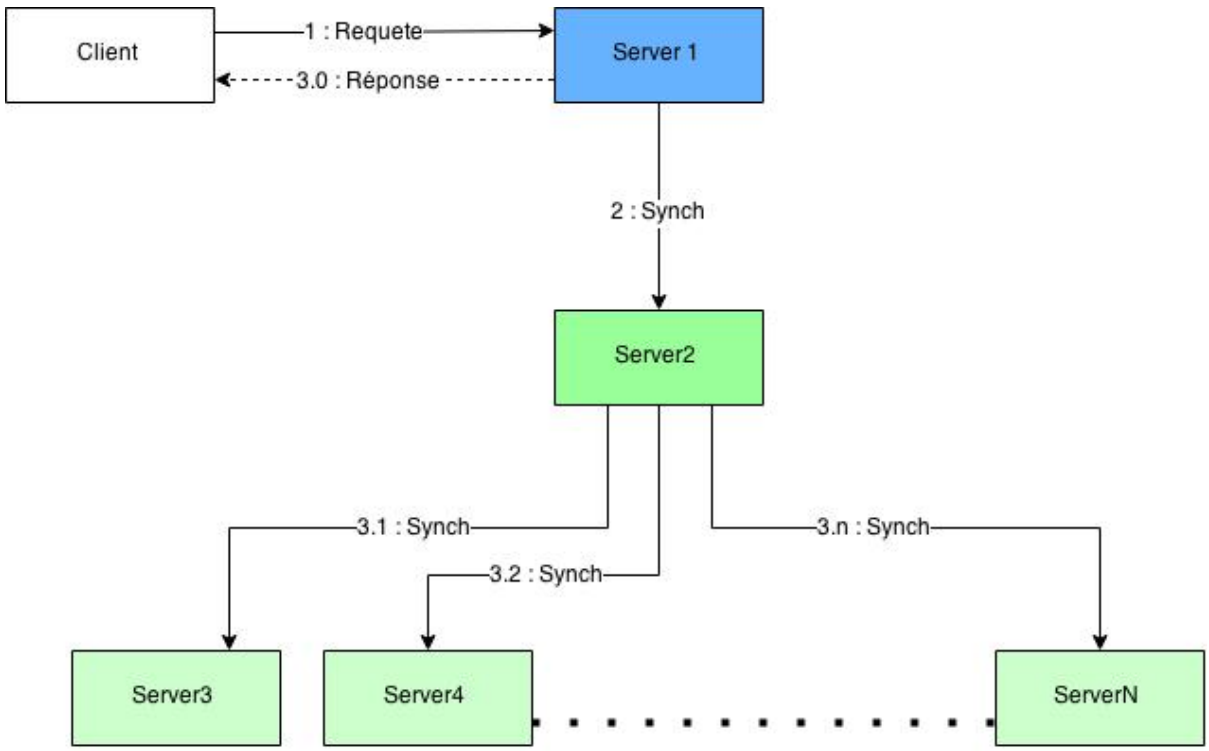
4.4 Diagramme synchro naïve



4.5 Diagramme arbre de synchro



4.6 Diagramme arbre agent



5 Conclusion

Ce laboratoire nous a permis de créer une petite architecture distribue sous la forme d'un client-serveur, avec plusieurs serveur agissant tel des backup si le serveur primaire ne répondait plus.
La synchronisation est la partie la plus demandante de ce processus avec différentes stratégie possible pour obtenir le même résultat mais avec des forces et des faiblesse différentes.
Nous nous somme contente d'implémenter la solution naïve, et d'identifier d'autres possible car le laboratoire ne semblait pas focaliser dessus.
L'ensemble des programme sont fonctionnel mais ne gère pas un grand nombre d'exception et d'erreur possible, du moins pas optimalement, ce qui devrait être le cas si l'on souhaite déployer ceci en production mais dans notre contexte la gestion est suffisante.
(e.g Nous n'avons pas géré par exemple si l'on n'avait plus de ressource pour créer un nouveau thread pour une nouvelle demande).

6 Annexes

Cette annexe présente les changements entres les différentes questions dans le code de façon normalisé.
Nous utilisons la commande linux : ‘diff -ur Qx Qy’ pour ceci. (Ou x et y représente le numéro des questions avec Qx le package associé.
Ces changements explicite sont disponible dans le dossier ‘doc’.