ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
Université du Québec

Cours: LOG736 - Fondements des systèmes distribués
Session: Automne 2024
Professeur: Marcos Dias de Assuncao
Département: LOGTI
Chargé de laboratoire: Thierry Fokou Toukam

Laboratoires - Projet 2
Paxos et Raft

1. Introduction

L'objectif de ce laboratoire est d'approfondir la compréhension des algorithmes de consensus Paxos de base et Raft. Vous réaliserez deux implémentations distinctes: la première reposant sur Paxos, avec une application inspirée de l'exemple d'organisation de fête vu en classe, et la seconde utilisant Raft pour répliquer des journaux (logs) entre plusieurs serveurs. Pour chaque algorithme, vous simulerez une panne et démontrerez la robustesse de l'application. Comme lors du travail précédent, vous utiliserez un langage de programmation tel que C, C++, Golang, Java, Python ou Rust, et vous ferez appel à la programmation par sockets.

Contrairement au précédent laboratoire, aucun code de départ n'est fourni. Vous devrez donc concevoir l'ensemble des implémentations.

2. Paxos

Dans cette partie, vous implémenterez une application dans laquelle plusieurs amis doivent se mettre d'accord sur la date et le lieu d'une fête. Chaque ami aura un droit de vote et participera au mécanisme de consensus. L'algorithme Paxos de base sera utilisé pour garantir la résilience aux pannes du système. Chaque ami jouera simultanément les rôles de **proposer**, **learner**, et **acceptor**. Vous devrez aussi implémenter un client pour interagir avec le système et proposer de nouvelles valeurs. Trois scénarios seront testés: panne d'un **acceptor**, d'un **proposer**, et d'un **learner**. Vous devrez démontrer que, malgré ces pannes, une date et un lieu sont toujours décidés.

(a) Question

Décrivez comment la propriété de vivacité peut être violée dans l'algorithme de Paxos.

(b) Implémentation du client

Le client interagira avec le **leader** des **proposers**, envoyant des requêtes d'écriture, et recevra les valeurs acceptées par les **learners**. Les méthodes à implémenter sont:

- **connectLeader:** établir une connexion avec le **leader.**
- **write:** soumettre une valeur (date/heure/lieu) au **leader**.
- **printReceivedValue:** afficher la valeur acceptée par les **learners**.

(c) Implémentation des amis

Les amis joueront les trois rôles principaux de Paxos. Voici les méthodes à implémenter:

Méthodes communes

- **connectAllPeers:** se connecter à tous les autres amis du réseau.
- **terminate:** simuler une panne en arrêtant un ami.

Méthodes proposer

Lors du démarrage d'un « ami », vous devez spécifier si celui-ci est un **leader** dans le réseau. Tous les **amis** dans le réseau doivent avoir connaissance de ce(s) **leader(s)**

- **sendPrepareRequest:** envoyer aux **acceptors** un numéro de proposition.
- **sendAcceptRequest:** envoyer une requête d'acceptation aux **acceptors** avec une valeur après avoir reçu un quorum de promesses.

Méthodes acceptor

- **sendPromiseResponse:** envoyer une promesse de respecter un numéro de proposition. Si l'**acceptor** a déjà promis à un autre **proposer** et qu'il a reçu un message **acceptedRequest**, il devra inclure dans sa réponse le numéro de la proposition actuelle ainsi que celui de l'ancienne promesse et sa valeur.
- **sendAcceptedRequest:** Cette méthode informe les **learners** que la valeur **v**, associée au numéro de proposition n, a été acceptée. Cela est vrai seulement si l'**acceptor** n'a pas déjà accepté une valeur avec un numéro de proposition supérieur à N. Dans le cas contraire, un message NACK avec le numéro de la plus grande proposition (la dernière promesse valide) est envoyé.

- **sendAcceptedResponse:** Cette méthode fonctionne comme **sendAcceptedRequest**, mais elle est envoyée au leader.

Méthode learner

- **sendClientResponse:** Cette méthode envoie la valeur décidée par consensus au client après avoir obtenu un quorum de réponses des **acceptors**.

(d) Évaluation

Votre implémentation devra répondre aux critères suivants:

- **Paxos 1:** Atteindre un consensus sur une valeur proposée par un client.
- **Paxos 2:** Une seule valeur peut être apprise par tour.
- **Paxos 3:** Tous les **learners** apprennent les valeurs.
- **Paxos 4:** Sûreté maintenue avec ***2f + 1*** processus et ***f*** pannes.
- **Paxos 5:** Respect du protocole de Paxos.
- **Implémentation 1:** Chaque **ami** est exécuté dans un processus unique.
- **Implémentation 2:** La communication se fait via des sockets en texte clair, et une connexion avec un client Telnet à n'importe quel processus doit permettre au client d'appeler les primitives du processus.

Vous devrez également fournir **cinq** tests unitaires non triviaux.

3. Raft

Vous implémenterez un automate répliqué en utilisant Raft. L'application distribuée devra traiter des opérations arithmétiques envoyées par un client au **leader**. Raft comporte deux phases: élection du **leader** et réplication des **logs**. Vous simulerez des pannes de réseau et de processus, et démontrerez la progression malgré ces pannes.

Raft est un protocole en deux phases : l'élection du **leader** et la réplication des **logs**. D'abord, un **leader** est élu, supervisé par une minuterie qui déclenche une nouvelle élection si aucun progrès n'est fait avant l'expiration du délai. Une fois élu, le **leader** transmet les commandes reçues des clients à tous les nœuds du réseau. Après avoir obtenu un quorum de réponses, il confirme l'opération et informe les nœuds que l'opération est désormais dans les **logs**.

Dans cet exercice, un client enverra des opérations au **leader**, qui les répliquera aux autres nœuds. Le client pourra également interroger le **leader** pour consulter l'état des **logs** à tout moment. Votre implémentation devra être résiliente aux pannes de réseau

(partitions) et de processus. Vous devrez implémenter l'élection du **leader**, la réplication des **logs**, et la réconciliation en cas de partition, tout en démontrant que l'implémentation progresse malgré ces pannes.

(a) Question

Décrivez les transitions d'états possibles d'un nœud dans Raft (**follower**, **candidate**, **leader**).

(b) Implémentation du client

Le client doit communiquer avec le(s) **leader(s)** du réseau pour introduire de nouvelles opérations dans le système. Après l'envoi d'une requête, une réponse contenant le résultat de l'opération sur l'état du système est renvoyée au client. Les opérations possibles sont **SET x**, **ADD x**, **SUB x**, et **MULT x**, où **x** est un entier. **SET x** définit la valeur du système à **x**, **ADD x** ajoute x à la valeur actuelle, **SUB x** soustrait **x**, et **MULT x** multiplie cette valeur **par x**. Le résultat de chaque opération est mémorisé par chaque nœud, et tous les nœuds doivent obtenir la même valeur si le protocole de consensus fonctionne correctement.

Les méthodes à implémenter sont:

- **connectLeader** : Établit une connexion avec un **leader** dans le réseau.
- **writeOperation** : Envoie une opération supportée par le protocole au **leader**.
- **getLogs** : Interroge un nœud pour obtenir la liste des opérations confirmées (**committed logs**).

(c) Implémentation des nœuds

Le code d'un nœud doit gérer toutes les transitions de rôle (**follower**, **candidate**, **leader**). Il doit implémenter la logique d'élection du **leader**, la réplication des **logs**, et la réconciliation des **logs** en cas de partition du réseau. Tous les nœuds doivent également exécuter les commandes des messages **appendEntries**. Les méthodes suivantes doivent être implémentées.

Méthodes communes

- **rollbackAndSynchronizeCommits**: Si un nœud ne peut accepter un message **appendEntries**, il doit synchroniser ses logs avec ceux du **leader**. Il doit revenir au point commun avec le **leader**, puis ajouter les entrées manquantes.
- **respondVote**: Cette méthode est appelée pendant l'élection du **leader** si le nœud n'a pas encore voté dans cette élection.

- **appendEntries**: Cette méthode est appelée périodiquement pour répliquer l'état des **logs** aux autres nœuds.

Méthode en état follower

- **acknowledge:** Cette méthode est appelée lorsqu'un nœud accepte une valeur proposée par le **leader** et envoie une réponse.

Méthode en état candidate

- **requestVote:** Cette méthode est appelée pendant l'élection du **leader** lorsqu'un follower souhaite se **proposer** comme **leader** auprès des autres nœuds.

Méthode en état leader

- **notifyClientCommit:** Cette méthode est appelée lorsque le **leader** reçoit un quorum de messages **acknowledge** des followers et envoie la valeur calculée au client.

(d) Évaluation

Afin d'évaluer l'implémentation, votre solution devra contenir ces spécifications fonctionnelles:

- **Raft 1:** Si un **follower** ne reçoit plus de messages d'un **leader** après un certain délai, il doit lancer l'élection en devenant **candidate**.
- **Raft 2:** Un **candidate** devient **leader** s'il obtient un quorum de votes des **followers**.
- **Raft 3:** Un quorum de messages **acknowledge** des **followers** actifs doit être reçu pour qu'une valeur soit confirmée dans les **logs** du **leader**.
- **Raft 4:** La durée de la minuterie d'élection doit être entre 100 et 300 millisecondes.
- **Raft 5:** Un seul client peut ajouter de nouvelles valeurs dans le système.
- **Raft 6:** Les logs du nouveau **leader** contiennent toutes les entrées confirmées par l'ancien **leader** (intégrité des données).

Vous devrez aussi fournir **cinq** tests unitaires non triviaux.

4. Évaluation du travail

Votre travail sera évalué selon les critères suivants:

- Réponse aux besoins fonctionnels, clarté et qualité du code (50%).
- Qualité des tests unitaires (20%).
- Réponse aux questions (20%).
- Démonstration (10%).