Raft Implementation Report

MEI - Tolerância a Faltas Distribuída - 2022/2023

Rafael Pilré - 59448

Tiago Silva - 59446

João Rolo - 59450

Tomás Ferreira - 59449

Índice

1. Introdução

No âmbito da disciplina de Tolerância a faltas distribuídas , foi proposta a implementação do algoritmo de consenso RAFT, criando a biblioteca cliente que é capaz de ligar o cliente ao serviço e a biblioteca servidor capaz de ligar vários servidores e executar um pedido feito pelo cliente, que também é responsável pela eleição de um líder forte.

A primeira parte do projeto pressupõe a implementação de RPC's.
Um dos RPCs é o *Invoke(oldMessage, msg, label,term,leaderId,lastlogIndex,commit)*, que é responsável por receber uma label ("ADD" ou "GET"), uma mensagem, o termo que irá se alterar assim que for elegido um novo leader, e um servidor, em seguida esse servidor, dependendo da label, irá retornar a String guardada ou adicionar uma String que ainda não esteja presente na mensagem.

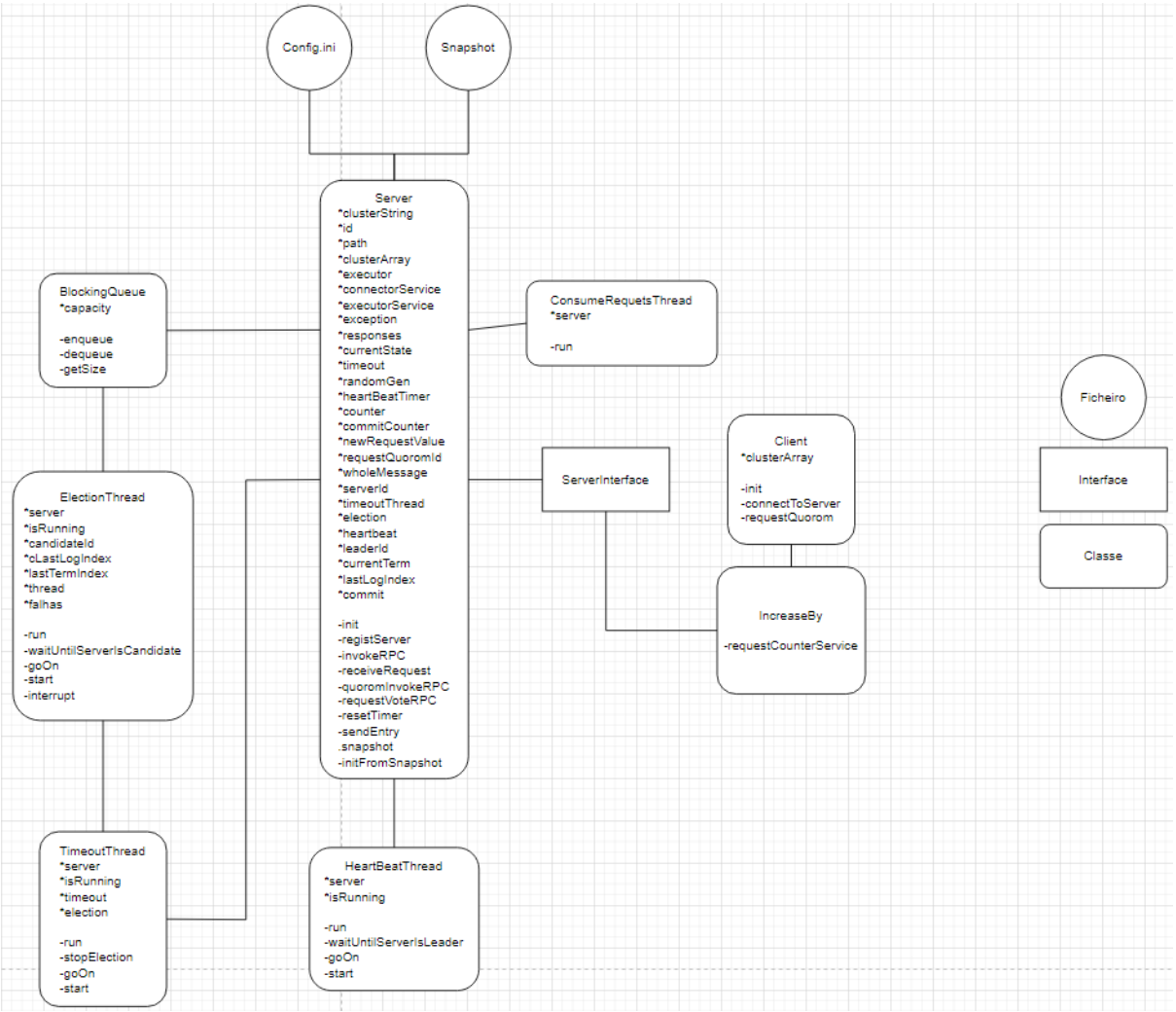
- O RPC *quorumInvoke*, é utilizado para invocar uma operação (request label, request data) em todas as réplicas menos o remetente, recolhendo as respostas da maioria (k>n/2-1) das réplicas.
- O *serverInterface* irá corresponder aos métodos que o cliente pode chamar que por sua vez irão referenciar o método implementado no servidor.//Podem adicionar isto na arquitetura

A segunda parte do projeto pressupõe a implementação da Leader Election. A ideia aqui é que todas as réplicas comecem como Follower e depois tentem ser o Leader através duma eleição até que alguma delas seja eleita.

- Quando o sistema começa, o Leader deve ser eleito de acordo com a eleição do mesmo.
- O Leader deve enviar de forma periódica heartbeats, de forma a informar as demais réplicas que ainda está vivo. As réplicas que são Follower deverão resetar os seus timers depois de receberem uma *AppendEntry(Invoke)* do leader.
- Se o tempo expirar, os Followers devem passar para Candidate e começar uma nova eleição, invocando o *RequestVote*.

Por fim, na terceira parte do projeto pressupõe a implementação do Log Replication, onde é implementada uma comunicação normal entre cliente e o Leader, ou seja, o cliente faz um pedido ao Leader, de seguida o Leader da *AppendEntry* para os Followers e o cliente recebe uma resposta do Leader e integra-a na Leader Election.

2. Arquitetura



3. Cliente

3.1 Como conectar/ler o ficheiro de configuração

O cliente começa por executar o método *init*, que será responsável por carregar o ficheiro de configuração, config.ini, que tem todos os endereços dos membros de cluster (ip e port), armazenando esses endereços na memória.

Em seguida, quando iniciamos um servidor especificamos qual o servidor que queremos abrir através do ip e port, registrando-o no *rmiregistry* logo a seguir.

O cliente vai executar a função *connectToServer*, onde temos um *clusterArray* com todos os ips dos servidores existentes. Aí iremos guardar na nossa variável *address* o endereço que está na posição dada. De seguida usamos a função *lookup*, de modo a encontrar o objeto pelo o seu nome, ou seja, encontrar o servidor com aquele endereço.

3.2 Request para o servidor

O cliente ao fazer um pedido para o servidor vai invocar o método *requestCounterService* da classe *increaseBy* a cada 9 segundos, incrementando valores aleatórios de 1 a 5. É depois nessa classe invocado o método *receiveRequest* que irá enviar o pedido do cliente para o servidor com a label e o contador como argumentos. No servidor, o método verifica que o pedido está a ser feito ao Leader e adiciona o pedido à queue de pedidos que posteriormente, utilizando a *ConsumeRequestThread* são dequeued e processados.

4. Servidor

4.1 Ler Init

O Servidor começa por executar a função *init* que tem a responsabilidade de ler o ficheiro de configuração e inicializar os atributos que são necessários para que o Servidor execute os pedidos que lhe serão tratados.

A função *init* começa por ler o ficheiro de propriedade que foi fornecido, inicializando-o depois de o fazer, o servidor prossegue e obtém toda a informação do ficheiro de propriedade que necessita:

- O seu endereço guardado em *id*, que é composto pelo seu próprio ip e o port que utilizará para a comunicação;
- Cria e preenche o *clusterArray*, um conjunto de endereços responsável por manter os endereços de todos os clusters da rede;

4.2 RequestVoteRPC

O método *RequestVoteRPC* é responsável por solicitar votos para um candidato quando este está a tentar tornar-se um Líder através de uma eleição. Quando este método é chamado, prossegue-se com a votação para verificar se este pode-se tornar-se um Líder, o que acontece quando tem a maioria de votos. Após a verificação, o mesmo prossegue com a implementação, o candidato invoca o método para tentar recolher votos de todos os outros followers, envia o *RequestVoteRPC* a cada follower e espera pelas suas respostas. Se os followers que recebem o *RequestVoteRPC* considerarem que o candidato tem os termos necessários para se tornar Líder, então ele define o atributo *VotedFor* no seu próprio estado para corresponder ao endereço de quem votou. Finalmente, depois de tudo isto ser feito, o temporizador é reiniciado através dum *quorumInvoke*, para reiniciar as *timeoutThreads* dos followers, invocado pelo recém-eleito líder.

4.3 HeartbeatThread

A *HeartBeatThread* é uma classe que estende uma *Thread*. Esta classe é inicializada e começa a funcionar no construtor do servidor. Quando a Thread começar a correr, estará num loop infinito que chamará a função auxiliar waitUntilServerIsLeader. Se o servidor for um follower ou um candidato, então ele aguardará por uma mensagem de notificação. Se o servidor for um líder, submeterá um *quorumInvoke* com a label ADD e uma String vazia. Depois de enviar o *HeartBeat* aos servidores, o processo dormirá durante um período de tempo definido. A mensagem de notificação para alertar o processo de que o servidor é um novo líder eleito é enviada chamando a função auxiliar de *goOn*. Esta função é chamada logo após o líder ter sido eleito.

4.4 ElectionThread

A election *Thread* inicia quando o estado do servidor passa a candidato, executando o *RequestVoteRPC* para as outras réplicas com o seu termo antigo e novo e o último logIndex retornando as respostas para a *blockingqueue*. Se algum servidor estiver em baixo, guardamos essa falha num contador e esperemos que haja uma maioria. Quando este processo ocorrer inicia-se uma nova *Thread* que vai dar *dequeue* à blocking queue e vai incrementar cada voto válido até que haja maioria. Quando esta for verificada o servidor passa para o estado de líder e aumenta o seu termo e executa os heartbeats.

4.5 SendEntry

SendEntry é um método que será executado quando ocorre um *quorumInvoke* e é despoletado quando uma réplica tem o seu log atrasado em comparação com o líder, invocando duas vezes o *InvokeRPC* com commit a false e true, o primeiro vai adicionar uma entry em falta no log do follower. O segundo *invokeRPC* vai com o commit a true, para que o state machine seja atualizado na réplica. Desta maneira iremos garantir sempre que os logs sejam replicados atualizando o state machine.

4.6 IncreaseBy

É um serviço de state machine que mantém um número inteiro inicializado com 0 e que fornece uma única operação que aumenta o valor do contador em x e devolve o seu valor. O *requestCounterService* fará esse serviço para todas as máquinas executando um *receiveRequest* com a label “ADD” e o contador x como argumentos sempre que é feito um pedido do cliente. No lado do cliente, feito o pedido, é criada uma nova thread que vai invocar o *requestCounterService* a cada nove segundos usando um incremento entre valores aleatórios de 1 a 5.

4.7 Timeout Thread

A Timeout Thread é um processo responsável pela verificação de se o Leader atual está vivo e se é necessário iniciar uma nova eleição. Sempre que uma réplica recebe uma mensagem do Leader, o seu timer leva reset, de modo a

continuar pela próxima mensagem. Se o timer chegar ao fim, inicia-se uma nova Leader Election e as réplicas mudam o seu estado para Candidate e tentam virar Leader.

4.8 BlockingQueue

BlockingQueue é um tipo de fila que permite operações seguras com threads. No RAFT é utilizada para facilitar a comunicação entre os diferentes nós num sistema distribuído. Quando um nó recebe uma nova entry, adiciona-a ao seu rlog e depois envia-a para os outros nós do sistema através da blocking queue. Os outros nós recebem a entry e anexam-na nos seus logs. Isto assegura que todos os nós têm uma visão consistente do registo, o que é necessário para que o algoritmo RAFT funcione corretamente.

4.9 ConsumeRequestsThread

Esta thread vai ser responsável por processar os pedidos do cliente. Aqui vai ser usada a blockingqueue para obter os pedidos do cliente e estes vão ser processados usando o método *quorumInvokeRPC*, passando a label e a data do pedido.

4.10 Comunicação um para um RMI (InvokeRPC)

O primeiro método verifica se o termo do líder é maior ou igual ao termo atual da réplica que recebeu o pedido. Se for, muda o seu estado para se tornar um Follower e redefine o seu timer e atualiza o seu termo. O método então processa o pedido dependendo do seu tipo (especificado pelo argumento label). Se a solicitação for "GET", o método simplesmente retorna o estado atual do log. Se a solicitação for uma solicitação "ADD", o método adiciona uma nova mensagem ao log e retorna o log atualizado.

O método também inclui alguma lógica adicional para lidar com os committed requests e manter a consistência. Por exemplo, se o pedido incluir o argumento commit definido como true, o método verificará se a última entrada no log do líder corresponde à última entrada no log, se corresponder, ele vai fazer uma computação, somando ao seu estado a entrada do log. Se eles não corresponderem, o método atualizará o log com o log do líder e definirá o commit

como verdadeiro. Isto garante que todos os nós no cluster tenham as mesmas entradas de log e estado e mantenham a consistência.

A cada dez commits é invocado o método snapshot para salvaguardar o estado da réplica e a última entrada no log.

4.11 Comunicação de um para muitos RMI (*quorumInvokeRPC*)

O método primeiro verifica se o servidor atual é o líder. Se não for, ele redireciona a solicitação para o líder. Se for o líder, ele cria uma nova thread para enviar o pedido para cada um dos outros servidores usando *Remote Method Invocation* (RMI) e aguarda uma resposta da maioria dos servidores antes de dar commit do pedido e retornar uma resposta ao cliente. Se a maioria dos servidores não puder ser alcançada, a solicitação não será confirmada.

4.12 receiveRequest

Neste método é feita uma verificação, de modo a que os pedidos sejam redirecionados para o Leader, caso estes sejam enviados a uma réplica que não seja o leader. Inicialmente é feita uma verificação para constatar se a réplica para qual os pedidos estão a ser enviados é o Leader. Se não estiverem é feito um *lookup* para redirecionar o pedido para o leader.

4.13 Snapshot

Snapshot é um método que vai ser executado no *InvokeRPC* a cada dez commits. Este método cria um ficheiro que irá incluir o último valor do log e o estado da state machine, preservando assim o estado da máquina. Se um dos servidores der crash, quando ele voltar irá usar o snapshot como ficheiro de inicialização.

Conclusão

Com este projeto conseguimos aplicar os conceitos de tolerância a falhas e perceber a sua importância num sistema distribuído. Considerámos que fizemos um bom trabalho na implementação do algoritmo de consenso RAFT, tendo conseguido aplicar com sucesso as ideias base do algoritmo de consenso, conseguindo eleger um líder forte e replicação de logs e máquinas de estado. Em geral, consideramos o Raft um algoritmo bom o suficiente para atingir o consenso numa rede síncrona, que para além da, relativamente "fácil" implementação, consegue aplicar os conceitos base de eleição de um líder forte e replicação de logs e máquinas de estado.