Raft

Miguel Alves, Nº 49828

1 Descrição de Alto Nível

O protocolo **Raft** é utilizado para replicar um log, garantindo o consenso sobre a ordem pela qual as diferentes entradas do mesmo serão aplicadas.

A cada participante no protocolo dá-se o nome de **peer**, e num determinado instante cada peer pode estar num de três estados, sendo estes **Follower**, **Candidate** e **Leader**.

Para além disso, cada **peer** guarda a seguinte informação:

- o `termo` em que se encontra: utilizado para saber o quão atualizado está um **peer**
- o `id` do último **peer** em quem votou
- os votos que recebeu na última eleição
- o seu `log` (sendo que cada entrada do log possui a operação correspondente, e o termo em que aquela operação foi adicionada ao log)
- o seu estado
- o índice da última entrada de log que foi `committed` (segundo o conhecimento que o **peer** possui)
- o índice da última entrada de log cuja operação foi executada pelo **peer**

1.1 Follower

Quando um peer se encontra no estado **Follower**, o seu comportamento baseia-se na receção de mensagens e no envio de resposta às mesmas.

Ao receber uma mensagem com uma operação vinda de um cliente, o **Follower** redireciona a mensagem para o líder.

Ao receber uma mensagem de um pedido de voto vinda de um **Candidate**, o **Follower** responde com o seu voto. O voto será positivo caso o termo e as entradas de log do candidato estejam atualizados, e caso o **Follower** ainda não tenha votado a favor de outro candidato. Caso contrário, o voto será negativo.

Ao receber uma mensagem de um pedido para adicionar entradas ao log vinda de um **Leader**, o **Follower** responde sempre com o seu termo atual, e com `TRUE` caso as entradas tenham sido adicionadas ao log, ou `FALSE` caso tenha havido algum conflito, caso o **Follower** tenha logs em atraso ou caso o **Leader** tenha um termo menor do que o **Follower**.

Caso o pedido de adicionar novas entradas ao seu log tenha sucesso, o **Follower** verifica também qual o `commit_index` do **Leader** (incluído na mensagem do pedido), e atualiza também o seu `commit_index`, aplicando as respetivas operações.

Um **Follower** altera o seu estado para **Candidate** quando existe um *heartbeat_timeout* (passou mais do que X tempo sem receber um pedido para adicionar entradas ao seu log).

1.2 Candidate

Quando um peer altera o seu estado para **Candidate**, é iniciado o processo de eleição. O **Candidate** começa por incrementar o seu termo atual, e envia um pedido de voto a todos os outros **peers**, contendo o seu termo atual, o índice da última entrada de log que possui e o respetivo termo associado à entrada.

Ao receber uma resposta negativa ao pedido de voto, o **Candidate** verifica se o termo incluído na resposta é maior do que o seu termo atual, e nesse caso o **Candidate** reverte o seu estado para **Follower** pois existe um **peer** mais atualizado que ele.

Ao receber uma maioria de respostas positivas ao pedido de voto, o **Candidate** altera o seu estado para **Leader**.

1.3 Leader

Para além da informação armazenada por todos os **peers**, um **Leader** guarda também a seguinte informação:

- para cada **peer**, o índice da próxima entrada que deve ser enviada para esse **peer**
- para cada **peer**, o índice da última entrada que sabemos que foi adicionada ao log desse **peer**

O **Leader** está encarregue de enviar mensagens de pedidos para adicionar entradas aos logs dos outros **peers** de X em X tempo, independentemente de haverem ou não novas entradas no seu log, pois estas mensagens servem também para informar os outros **peers** de que o **Leader** não falhou.

Ao receber uma resposta a um pedido que não teve sucesso, decrementa o índice da próxima entrada de log que deve ser enviada ao **peer** correspondente. No pior cenário, o conflito encontra-se no ínicio do log do **peer** e o pedido só terá sucesso quando enviarmos as entradas todas desde o início do log.

Sempre que recebe uma resposta a um pedido que teve sucesso, o **Leader** verifica qual é a última entrada a que pode dar `commit`, que será a entrada com maior índice que se encontre replicada numa maioria dos **peers**.

Ao receber qualquer mensagem que contenha um termo maior que o seu, o **Leader** reverte o seu estado para **Follower**, pois isso significa que haverá um outro **Leader** mais atualizado que ele.

2 Estrutura das Mensagens Trocadas entre Peers

Segue-se a estrutura base das mensagens enviadas. De notar que consoante as implementações e/ou linguagens, as mensagens podem conter mais informação.

- `RequestOperation` → operação que deve ser replicada
- `RequestVote` → termo do candidato; id do candidato; índice da última entrada no log do candidato; termo da última entrada no log do candidato
- `ResponseVote` → termo do seguidor; id do follower; `TRUE` se o follower votou no candidato, `FALSE` caso contrário
- `RequestAppend` → termo do líder; id do líder; índice da última entrada de log que o líder acha que o seguidor tem; termo da última entrada de log que o líder acha que o seguidor tem; lista de entradas que o seguidor deve adicionar ao seu log; índice da última entrada `committed` pelo líder
- `ResponseAppend` → termo do seguidor; id do seguidor; `TRUE` se o seguidor adicionou as entradas ao seu log, `FALSE` caso contrário; o índice da última entrada no log do seguidor

3 Etapas do Protocolo

Podemos assim definir os passos/etapas de um possível fluxo de execução deste protocolo da seguinte forma:

1. Ocorre um `heartbeat_timeout` num **peer**, mudando o seu estado para **Candidate**
2. O **Candidate** inicia uma eleição:
   (a) incrementa o seu `termo`
   (b) vota nele próprio
   (c) envia uma mensagem de `RequestVote` a todos os outros **peers**
3. Cada **peer**, ao receber a mensagem de `RequestVote`:
   (a) se o termo e log do **Candidate** estão atualizados, e ainda não votou em nenhum outro candidato, responde com uma mensagem `ResponseVote` com valor `TRUE`
   (b) caso contrário, responde com uma mensagem `ResponseVote` com valor `FALSE`
4. O **Candidate**, ao receber uma maioria de votos com valor `TRUE`, altera o seu estado para **Leader**
5. O **Leader** envia uma mensagem de `RequestAppend` para todos os **peers**
6. Cada **peer**, ao receber a mensagem de `RequestAppend`:
   (a) se o `termo` do **Leader** for menor que o seu, responde com `ResponseAppend` com valor `FALSE`
   (b) se o seu termo for menor que o do **Leader**, atualiza o seu termo
   (c) se o índice da última mensagem de log que o **Leader** acha que o **peer** tem for maior que o atual, então responde com `ResponseAppend` com valor `FALSE` pois o **peer** tem entradas em atraso
   (d) se o termo da última entrada do log do **peer** for diferente do que o **Leader** tem conhecimento, então remove todas as entradas a partir desse índice do log, e responde com `ResponseAppend` com valor `FALSE` pois o **peer** possuía um conflito no log
   (e) se todas as verificações anteriores passarem, então o **peer** adiciona as entradas ao seu log, atualiza o `commit_index`, aplica operações `committed` que ainda não tenha aplicado, e responde com `ResponseAppend` com valor `TRUE`
7. Ao receber uma mensagem de `ResponseAppend` de um **peer**, o **Leader**:
   (a) se a mensagem possui o valor `FALSE`, decrementa o índice da próxima entrada que deve ser enviada a esse **peer**
   (b) atualiza índice da próxima entrada que deve ser enviada para esse **peer**
   (c) atualiza o índice da última entrada que sabemos que foi adicionada ao log desse **peer**
   (d) verifica se pode efetuar `commit` a uma nova entrada (se a entrada em `commit_index+1` se encontra na maioria dos logs dos **peers**)
   (e) verifica se tem alguma entrada `comitted` por aplicar, e em caso afirmativo aplica-a

4 Implementação em Rust

Para a implementação em **Rust**, foram feitas as seguintes escolhas de implementação:

- Cada agente é simulado por uma thread
- A troca de mensagens é feita com recurso ao módulo std::sync::mpsc
- Adicionou-se uma probabilidade (ajustável) de uma mensagem não ser enviada
- Adicionou-se uma probabilidade de um peer falhar

O mapeamento entre as etapas do protocolo e as funções em Rust é o seguinte (usando a mesma numeração que em ***3***):

1. Na função `run`, a execução da função `timed_out` devolve `TRUE`, levando à chamada da função `begin_election`
2. Execução da função `begin_election`
3. Um **peer** recebe a mensagem de `RequestVote` o que leva à execução da função `handle_vote_request`
4. Um **Candidate** ao receber a mensagem de `ResponseVote`, executa a função `handle_vote_response`, que por sua vez leva à execução da função `become_leader`
5. O **Leader** executa a função `send_entries`, que por sua vez executa a função `append_entries` para cada um dos **peers**
6. Cada **peer**, ao receber a mensagem de `RequestAppend` executa a função `handle_append_entries_request`
7. Ao receber cada mensagem de `ResponseAppend`, o **Leader** executa a função `handle_append_entries_response`, que por sua vez chama as funções `update_commit_index` e `apply_new_commits`

5 Testes e Resultados

// **TODO** (quais os testes realizados; comparação de tempos de execução com o aumento do número de agentes e aumento do número de propostas concorrentes; comportamento perante falhas; etc.)

6 Fontes

In Search of an Understandable Consensus Algorithm (Extended Version), Diego Ongaro & John Ousterhout, https://raft.github.io/raft.pdf

Formal TLA+ specification for the Raft consensus algorithm, Diego Ongaro, https://github.com/ongardie/raft.tla