Tolerância a Faltas
Improving Reading Scalability in Raft

Guilherme Fernandes[PG50407], Matilde Bravo[PG50651], and Mariana Rodrigues[PG50622]

Universidade do Minho

1 Introdução

Raft é um algoritmo de consenso em sistemas distribuídos, utilizado para gerir réplicas de um *log*. Desta forma é possível que um conjunto de máquinas (nós) se comportem como um grupo coerente, capaz de suportar falhas de alguns dos seus membros sem comprometer o *log*. Por forma a garantir *strong consistency*, todos os pedidos de leitura e escrita provenientes do cliente devem passar pelo *leader*. Os restantes nodos serão utilizados apenas como "*cold standbys*", nunca respondendo ao cliente diretamente. Contudo, isto representa um problema a nível da escalabilidade e uma falta de aproveitamento dos recursos disponíveis. Desta forma, o *leader* é sobrecarregado com pedidos, enquanto que os *followers* apenas replicam os valores por ele enviado.

Como forma de melhorar a escalabilidade, o artigo *"Leader or Majority: Why have one when you can have both? Improving Read Scalability in Raft-like consensus protocols"* propõe uma variação do protocolo **Raft** que permite que outras réplicas, além do líder, possam responder a pedidos de leitura em certas condições, distribuindo assim a carga de trabalho. Esta distribuição é possível devido ao facto de um valor *commited* estar obrigatoriamente replicado numa maioria dos servidores. Consequentemente, ao realizar uma leitura de uma maioria garante-se que este estará presente.

Com este trabalho pretende-se explorar a eficácia da estratégia proposta, comparando-a com o protocolo **Raft** padrão, utilizando a *framework Maelstrom* para simular um sistema distribuído.

2 Implementação

Começou-se por implementar uma versão base de *raft* e obter as devidas métricas para depois se proceder à implementação da nova versão proposta.

2.1 Raft

Foram criadas várias classes correspondentes aos vários estados possíveis dos nodos: *leader*, *follower* e *candidate*. A classe principal designa-se por `Node` e contém todo o estado e lógica comum aos vários estados, incluindo o principal *handler* de mensagens. No geral, o estado guardado nas classes corresponde ao que é sugerido pelo *paper* do **Raft**, apenas tendo sido acrescentadas variáveis consideradas úteis, como por exemplo, estado necessário para o *maelstrom* e um *timer* para os *timeouts*. Todas as mensagens têm um *handler* correspondente que irá devolver sempre o estado resultante de processar a mensagem, o qual poderá ser o mesmo ou um novo.

Ao implementar o *AppendEntries RPC*, deparamo-nos com um problema que se deve ao facto de estarmos a fazer uma implementação baseada em eventos. Quando o *leader* envia a lista de *entries* aos *followers* continua a processar outras mensagens e poderá acrescentar mais *entries* ao *log* antes de receber uma resposta destes. Quando recebe a resposta terá que incrementar o `next_index` e o `match_index` mas, apenas com os parâmetros sugeridos, não saberá quantas entradas tinha enviado e, logo, não saberá em quanto incrementar os índices. Por este motivo, nesta implementação, os *followers* respondem com um campo adicional, designado por *last_index* que corresponde ao índice do ultimo elemento do *log*.

Uma das questões cruciais em sistemas baseados em consenso é a *linearizability*. Existem alguns cenários em que o cliente poderia receber uma resposta com *stale data*, pelo que é necessário tomar precauções para evitar esta problemática. Um dos cenários ocorre quando um nodo recebe um pedido

de *read* e não tem conhecimento que já não é *leader*. Por esse motivo, é necessário garantir que ainda é *leader* antes de responder a um pedido *read-only*. Uma solução possível é forçar o nodo a enviar *heartbeats* e esperar por uma maioria, antes de responder ao pedido. Se já não for *leader*, então, numa maioria de nodos, pelo menos um responderá com o termo atualizado e o nodo converter-se-á a *follower*. O outro cenário em que poderá devolver *stale data* corresponde ao caso em que o *leader* não tem conhecimento de que uma ou mais entradas já foram *committed*. Apenas terá conhecimento assim que realizar *commit* de uma nova entrada, pois as anteriores também o serão por estarem replicadas numa maioria. Uma potencial sugestão dada pelo *paper* passa por realizar um *commit* de uma entrada "no-op" assim que inicia o mandato.

Nesta implementação, optamos por adicionar os pedidos *read* ao *log*, o que já nos permite resolver ambas as problemáticas. Quando um nodo tentar fazer *commit* do pedido *read*, se já não for *leader* e julgar sê-lo, irá obter essa informação quando espera resposta de uma maioria e também saberá quais são as entradas *commited* assim que tentar fazer *commit* desta.

2.2 Raft com Quorum Reads

Para esta implementação foi criada a classe `GatewayNode` que inclui um nodo *Raft*(camada abaixo) e o estado necessário para realizar os pedidos de leitura. Os pedidos de *write* e *cas* provenientes do cliente, bem como todos os tipos de mensagens do **Raft**, serão passados à camada de baixo, sendo invocado o respetivo *handler*. Os pedidos de *read* e as novas mensagens criadas para os processar serão tratadas por esta camada de cima. Quando um pedido de *read* é recebido, no caso de se tratar de um *leaseholder*, este realiza uma leitura direta. Se não se tratar de um *leaseholder*, então se será feita uma decisão, com base na probabilidade descrita no *paper*, se será feito um pedido ao *leaseholder* ou será feito um *quorum read*. Um *quorum read* implica selecionar uma maioria de nodos aleatoriamente, enviar mensagens do tipo *quorum_read* e aguardar pela resposta de todos (*quorum_read_response*). Foram criadas várias classes para armazenar o estado necessário, mantendo registo das mensagens já recebidas, bem como o maior *timestamp* e o seu valor correspondente. Uma vez que é necessária a receção de resposta de todos os nodos aos quais se enviou pedido, caso um destes não responda, optamos por fazer *timeout*.

Quando a decisão tomada é de fazer uma leitura ao *leaseholder*, obtem-se o id do *leader* invocando um método do Raft e, de seguida, é enviada uma mensagem do tipo *leaseholder_read*.

Na nossa implementação, o *leaseholder* corresponde ao lider do **Raft**, podendo este nodo fazer leituras diretas à base de dados. Como mencionado, quando um *follower* recebe um pedido de leitura, este tem uma probabilidade de fazer um pedido de leitura diretamente ao *leaseholder* (*leaseholder_read*), que, por sua vez, responde ao nodo que realizou o pedido com uma *leaseholder_response* que contém informação da leitura, este, por sua vez, responde ao cliente, com uma mensagem *read_ok* ou um erro caso a chave não exista na base de dados ou não exista um *leaseholder* (neste caso poderia ser feita uma nova tentativa, no entanto, optamos por falhar, e a responsabilidade de voltar a tentar é do cliente).

Leituras a *leaseholders* têm uma pequena chance de falhar se o líder não estiver atualizado, o que pode acontecer quando existem dois líderes em simultâneo. Isto poderia ser corrigido adicionando um *timer* que seria atualizado sempre que o líder recebe uma maioria de respostas a heartbeats ou quando entradas do *log* são aplicadas. De modo a garantir que só existe um *leaseholder*, mesmo havendo mais do que um líder, o valor de *timeout* teria de ser inferior ao *election timeout*.

3 Exemplos

3.1 Leitura direta é não-linearizável

Uma vez que os nodos *follower*s só são informados que um comando foi *committed* na próxima chamada do RPC *AppendEntries*, uma leitura de um nodo *follower*, imediatamente após uma escrita, não é linearizável.

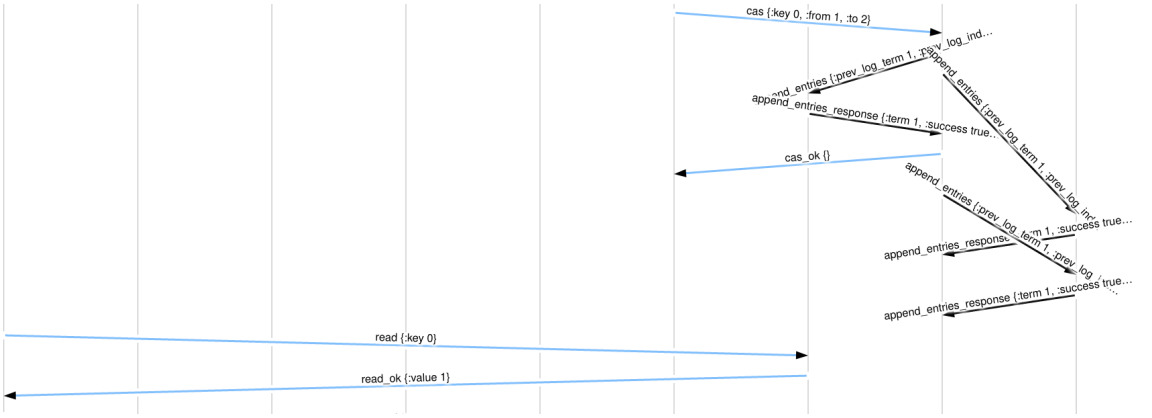

Fig. 1. Leitura direta é não-linearizável

Além disso, uma leitura direta num lider também pode ser não-linearizável, no caso de existirem mais do que um líder num dado intervalo de tempo.

3.2 Mudança de líder

De forma a ser possível visualizarmos o funcionamento da mudança de líder que foi implementado no protocolo de *Raft*, foi usado o auxílio do *Maelstrom* para a obtenção das seguintes imagens.

Como se pode observar, inicialmente o líder era o servidor n2.

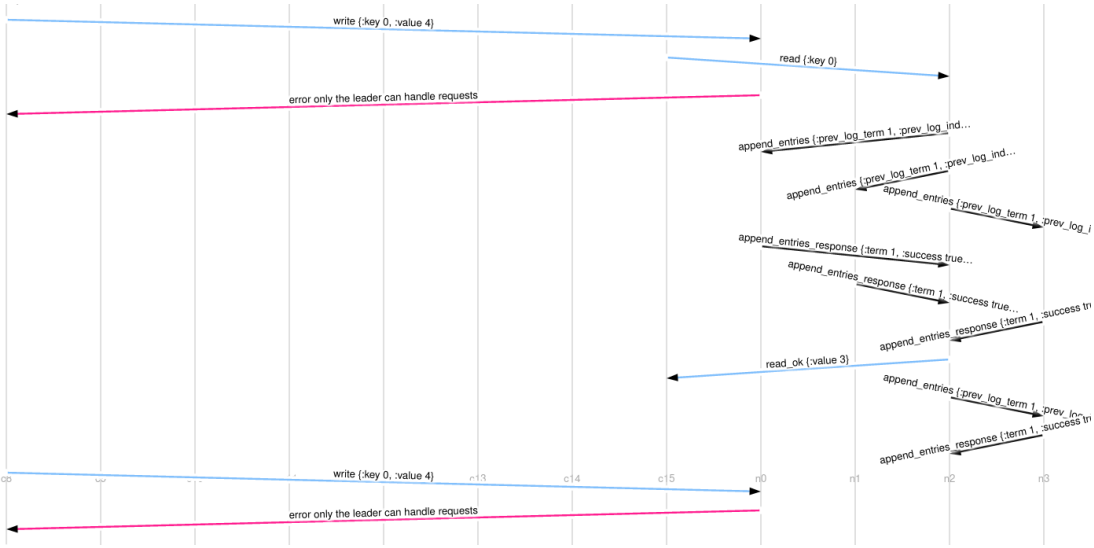

Fig. 2. Servidor n2 como Líder

Porém, para efeitos de demonstração, terminou-se esse nó após um pequeno intervalo de tempo. Não enviando nenhum *heartbeat*.

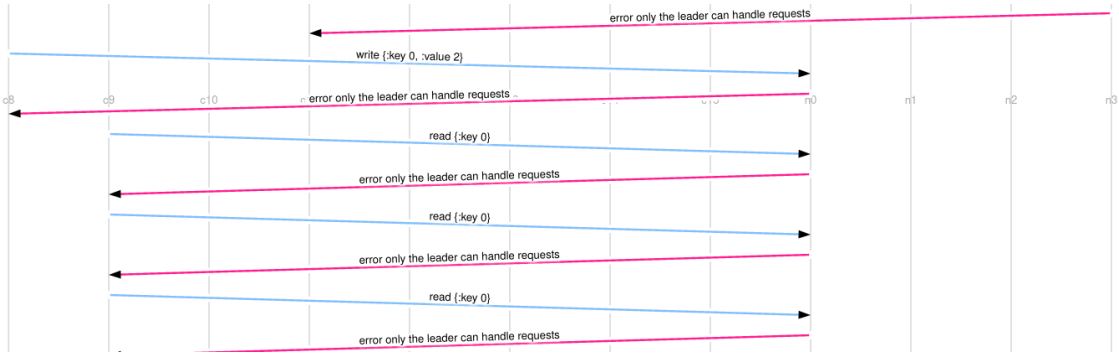

Fig. 3. Morte do Líder

Visualizando a seguinte imagem, observa-se que o servidor n1 deu *timeout* e tornou-se num candidato, realizando pedidos de *request_vote* aos restantes servidores.

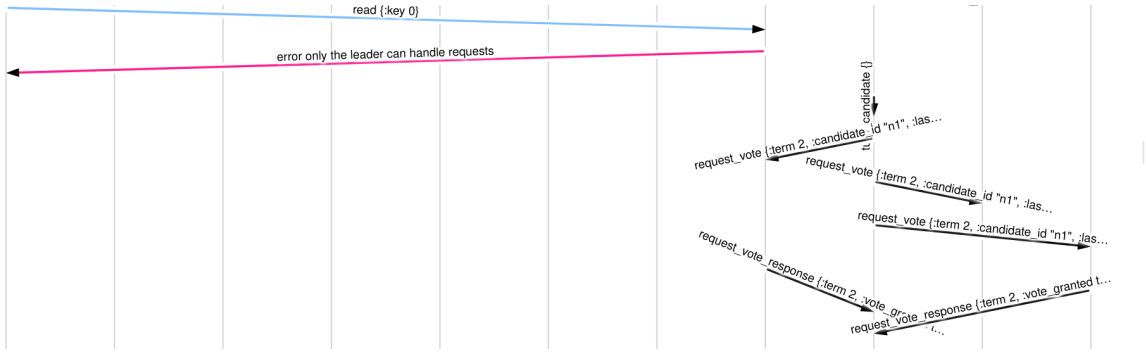

Fig. 4. Eleição do novo Líder

Logo adiante, este recebe a confirmação da maioria e passa a ser o líder.

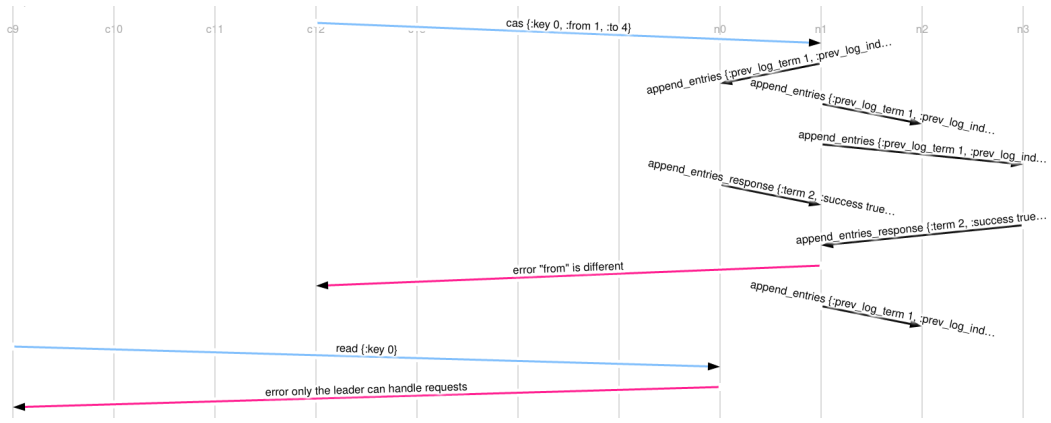

Fig. 5. Servidor n1 como Líder

4 Conclusão

Através das experiências realizadas neste trabalho, podemos concluir que a variante proposta do protocolo **Raft** apresenta melhorias significativas na escalabilidade de leituras em comparação com o protocolo **Raft** padrão.

Foi possível observar que, a distribuição de leituras entre várias réplicas, em certas condições, resultou num melhor desempenho do sistema, reduzindo o tempo necessário para processar solicitações de leitura. Naturalmente, também se observou uma redução na quantidade de erros provenientes de contactar um nodo incapaz de responder ao pedido.

Concluindo, sentimos que a pesquisa e realização deste trabalho contribuíram para um melhor entendimento do protocolo **Raft** e as suas variantes, demonstrando a importância de avaliar o desempenho de sistemas distribuídos sob diferentes cenários.

References

1. Leader or Majority: Why have one when you can have both? Improving Read Scalability in Raft-like consensus protocols
   `https://www.usenix.org/system/files/conference/hotcloud17/hotcloud17-paper-arora.pdf`
2. In Search of an Understandable Consensus Algorithm (Extended Version)
   Diego Ongaro and John Ousterhout
   Stanford University
   `https://raft.github.io/raft.pdf`
3. The Raft Consensus Algrothm, site visitado a 12-05-2023
   `https://raft.github.io/`
4. Maelstrom
   `https://github.com/jepsen-io/maelstrom`