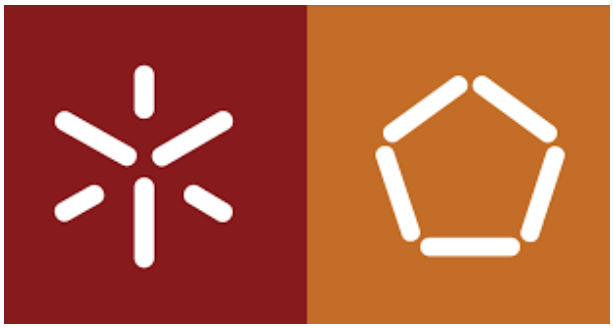

Universidade do Minho

Mestrado em Engenharia Informática
Tolerância a Faltas

Trabalho Prático

Diana Rodrigues
pg50320

José Fernandes
pg50525

Mariana Amorim
pg50623

Miguel Fernandes
pg50654

12 de maio de 2023

Conteúdo

1 Introdução

Este trabalho tinha como objetivo implementar uma variante do protocolo de consenso Raft em que a carga de trabalho está mais distribuída pelos vários servidores. No Raft apenas o *leader* responde a pedidos de leitura enquanto que nesta variante todos os servidores podem responder a esses pedidos.

Ao longo do relatório será descrito o protocolo assim como as decisões que foram sendo tomadas à medida em que o mesmo foi sendo implementado. Começamos por descrever as duas fases em que se divide o protocolo Raft: *leader election* e *log replication*. De seguida, descrevemos os pormenores relacionados com a variante do protocolo (*read quorum*), nomeadamente, a forma como escolhemos aplicar uma política de *back-off*.

Por fim, apresentamos os resultados de alguns testes feitos mais especificamente que mostram a mudança do líder e que uma leitura direta não é linearizável. Para além disso é ainda apresentada uma comparação entre o desempenho do Raft e desta variante.

2 Raft: Leader Election

O processo de (re)eleição de um *leader* no protocolo Raft inicia-se em duas ocasiões:

- Quando um servidor está à espera de receber um *heartbeat* (**wait_for_heartbeat**) e o *timer* **heartbeat_timeout** termina sem que o tenha recebido. O que significa que não existe um *leader* ou que a mensagem não chegou por atraso ou porque o servidor do *leader* parou.
- Quando um servidor é candidato num termo mas após um dado tempo (**election_timeout**) não conseguiu obter a maioria dos votos. Isto pode indicar que houve um empate no termo em questão, neste caso é reiniciado o processo de eleição para evitar *deadlocks*.

Nessas ocasiões é, então, invocado o método **request_vote**: o servidor torna-se *candidate* de um novo termo, vota em si próprio e envia uma mensagem do tipo *request_vote* para os restantes servidores. Para além disso, dá início ao *timer* **election_timeout**.

Quando um servidor recebe um pedido do tipo *request_vote* de um candidato pode inserir-se em uma destas três situações:

- o termo do candidato é superior ao termo em que o servidor se encontra;
- os termos de ambos os servidores são iguais, no entanto o servidor que recebe o pedido ainda não votou neste termo ou já votou no candidato que lhe enviou o pedido em questão;
- os restantes casos: o termo do candidato é inferior, o termo é o mesmo mas o servidor já votou noutro candidato ou o candidato tem o **log** menos atualizado.

Na primeira situação o servidor que recebe o pedido torna-se *follower* e atualiza o seu termo (método **become_follower**). Para além disso, se este servidor era anteriormente um *follower* deve cancelar o *timer* de espera por um *heartbeat*, pois o antigo *leader* (de que estava a receber os *hearbeats*) não deverá enviar mais mensagens desse tipo. Por outro lado, se antes era um candidato então deve cancelar o *timer* que evita os empates nas eleições pois o termo no qual ele era candidato já foi dado como terminado.

Nesta situação o servidor vota no candidato a menos que o mesmo esteja menos atualizado no que diz respeito às entradas no **log** (linhas 6 e 7).

```
1  if (msg.body.term > currentTerm):
2      become_follower(msg)
3      if (msg.body.lastLogIndex > len(log)-1):
4          votedFor = msg.body.candidate_id
5          reply(msg, type='request_vote_resp', vote_granted=True, term=currentTerm)
6      else:
7          reply(msg, type='request_vote_resp', vote_granted=False, term=currentTerm)
```

Na segunda situação, como o candidato está no mesmo termo e ainda não votamos em ninguém ou já votamos neste mesmo candidato, votamos neste candidato (ou voltamos a votar) a não ser que o **log** dele esteja menos atualizado, analogamente à situação anterior.

```
if (msg.body.term == currentTerm) and
    (votedFor == None or votedFor == msg.body.candidate_id) and
    (msg.body.lastLogIndex > len(log)-1):
        votedFor = msg.body.candidate_id
        reply(msg, type='request_vote_resp', vote_granted=True, term=currentTerm)

```

Por fim, se o candidato estiver num termo anterior, tiver o **log** menos atualizado ou se o servidor já tiver votado num outro candidato então não será concedido o voto a este candidato.

```
else:
    reply(msg, type='request_vote_resp', vote_granted=False, term=currentTerm)
```

Após avaliar em que situação se encontra e agir em conformidade, se o servidor não for *leader* então irá ficar à espera de um *heartbeat* que o irá informar da existência de um novo *leader*.

Por sua vez, os candidatos quando recebem votos (*vote_granted* = *True*) devem verificar se a mensagem é relativa ao termo atual, de modo a garantir que não é uma mensagem de termos anteriores que chegou atrasada. Feita esta verificação, deve aumentar o seu contador de votos e, caso o mesmo já ultrapasse uma maioria dos servidores, então o servidor afirma-se como *leader* e começa a enviar *heartbeats* para os restantes servidores.

No entanto, se a resposta recebida corresponder a um termo mais avançado, o servidor deve tornar-se *follower* (**become_follower**) e esperar por *heartbeats* do *leader*.

3 Raft: Log Replication

Depois de eleito, o *leader* fica então disponível para processar pedidos dos clientes. Neste capítulo abordaremos com mais detalhe a gestão e resposta a pedidos do tipo *write* e *cas*.

Após a receção de um pedido *write*, verifica-se se o processo de eleição já terminou e o estado do servidor em questão, isto é, se é *leader* ou não. Se o processo de eleição ainda não tiver terminado, ou seja, ainda não existir um *leader*, a resposta ao pedido é uma mensagem de erro. Caso contrário, e se o servidor que recebeu o pedido não for o *leader* eleito, o pedido é reencaminhado para o mesmo. Finalmente, se já existe um *leader* eleito e é ele que recebe o pedido, é invocado o método **create_command**.

O método **create_command** é invocado apenas pelo *leader* e serve, essencialmente, para este adicionar comandos/pedidos à sua lista **log**. Para além disso, é criada uma variável **index**, que corresponde ao tamanho da lista **log** já contando com a inserção da nova entrada, e atualizada a variável **matchIndex**, na posição relativa ao *leader*, com o valor de **index**.

A nova entrada da lista **log** trata-se de um objeto da classe **Command**. Esta classe tem como variáveis o tipo da mensagem (*write* ou *cas*), o valor da chave afetada, o valor que se pretende atribuir à chave em questão, o termo atual, a variável ***index*** mencionada anteriormente e a mensagem que o cliente enviou com o pedido.

3.1 Envio de mensagens *log_replication*

Como mencionado no capítulo anterior, o processo de eleição de um *leader* termina quando se verifica uma maioria de votos. Aquando desta verificação da maioria e consequente confirmação de *leader*, é criada e inicializada uma **Thread** com *target* para o método **send_heartbeat**.

O método **send_heartbeat**, por sua vez, é utilizado enquanto o servidor em questão é o *leader* e tem como principal função o envio de mensagens do tipo *log_replication*. Estas mensagens podem representar simplesmente um *heartbeat*, ou seja, uma mensagem do tipo *log_replication* com o campo *entries* vazio, ou com conteúdo da lista **log**, mais concretamente, valores que ainda não tenham sido enviados a servidores específicos.

O envio de mensagens deste tipo é efetuado de acordo com o valor de *sleep* definido nos 70 milissegundos. Para a escolha do valor em causa, o grupo teve em conta o *heartbeat_timeout* mencionado no capítulo anterior, que, sendo definido com um valor aleatório entre 150 a 300 milissegundos, obriga a que o intervalo de tempo entre o envio de cada *log_replication* seja inferior ao valor mínimo possível de *heartbeat_timeout*, ou seja, 150 milissegundos. Para além disso, foi necessário considerar possíveis atrasos no envio das mensagens.

3.2 Receção de mensagens *log_replication*

Quando um servidor recebe uma mensagem do tipo *log_replication* é invocado o método **verify_heartbeat**. Este método define se existe ou não alguma incompatibilidade entre a informação que recetor da mensagens detém e a informação acerca da replicação do *log* proveniente da mensagem.

O primeiro aspeto a ser verificado é o termo incluído na mensagem. Esta verificação é feita com recurso ao método **verify_term**. Caso o termo da mensagem seja menor do que o termo atual é retornado *false*; caso o termo da mensagem seja igual ao termo atual do servidor, cancela-se o *heartbeat_timeout* e é retornado *true*; caso o termo da mensagem seja superior ao atual, o servidor altera o seu estado para *follower*, atualiza o seu termo é retornado *true*.

Passada a verificação do termo, é necessária a reinicialização do *heartbeat_timeout* sinalizando a receção de uma mensagem *log_replication* válida, ou seja, que não exige nova eleição.

De seguida é feita a verificação das entradas da lista **log** que devem ser replicadas e, eventualmente, atualizadas variáveis de controlo como **commitIndex** e **lastApplied**. É também nesta fase que o dicionário **linkv** é atualizado, quando necessário.

Por fim, com base no valor retornado pelo método **verify_heartbeat** é enviada para o líder uma mensagem do tipo *log_replication_resp*, cujo campo *success* coincide com o valor mencionado anteriormente. Caso esse campo tome o valor de *true* é ainda adicionado o campo *term* contendo o valor do termo atual e o campo *matchIndex* contendo o valor do tamanho da lista **log**(utilizado para saber as entradas confirmadas); caso o campo *success* tome o valor de *false*, é adicionado à mensagem a enviar apenas o termo correspondente ao da mensagem recebida.

3.3 Receção de mensagens *log_replication_resp*

O processo de replicação de *log* termina com a avaliação de mensagens do tipo *log_replication_resp* por parte do *leader*. Inicialmente, é verificado o campo *success* da mensagem recebida. Como mencionado no subcapítulo anterior, existem duas formas desse campo apresentar o valor *false*.

A primeira acontece quando o termo correspondente à mensagem é superior ao termo atual do *leader*, ou seja, o *leader* apercebe-se que algum servidor se encontra num termo superior e, portanto, deve abandonar a posição de *leader*. A segunda acontece quando existe um conflito entre as entradas de *log* que realmente foram replicadas pelo servidor que enviou a mensagem de resposta e as entradas de *log* que o *leader* assume que o mesmo servidor replicou. Isto é, existe a necessidade de reenviar conteúdo para o servidor em causa. Para isso o indíce correspondente a esse servidor na variável **nextIndex** é decrementado em uma unidade até que encontrem uma entrada em comum.

Por outro lado, se o campo *success* da mensagem recebida toma o valor de *true*, cabe ao líder avaliar se recebeu pelo menos uma maioria de respostas em relação à mensagem *log_replication* em causa. Nesse caso, é então atualizado o seu dicionário **linkv**, são incrementadas variáveis de controlo como **lastApplied** e **commitIndex** e é enviada uma mensagem de confirmação *write_ok* ou *cas_ok* consoante o pedido.

4 Variante Raft: Read Quorum

Nos capítulos anteriores, foi descrita detalhadamente a implementação do protocolo **Raft**, que apresenta uma clara limitação quanto à distribuição da carga de trabalho. No artigo "*Leader or Majority: Why have one when you can have both? Improving Read Scalability in Raft-like consensus protocols*" é apresentada uma solução para este problema, esta variante do **Raft** permite que os pedidos de leitura sejam respondidos por qualquer servidor, reduzindo assim a carga de trabalho do *leader*.

Uma vez que o protocolo **Raft** não garante que todos os servidores têm a versão mais recente de uma chave, os pedidos de leitura a servidores que não são o *leader* poderiam originar respostas desatualizadas ou até mesmo erradas. Dessa forma, e tal como descrito no artigo anteriormente referenciado, é necessária a utilização de *quorums* de leitura.

Para aliviar a carga de trabalho do *leader*, não é lógico incluí-lo nos *quorums* de leitura, o que pode gerar um novo problema quanto à consistência das respostas. Isto acontece porque um servidor pode ter uma entrada no seu *log* referente a uma chave de leitura que já foi confirmada pelo *leader*, mas que o servidor ainda não recebeu confirmação. Para evitar este problema foi implementado o algoritmo "*Strongly Consistent Quorum Reads*", também descrito no artigo:

- As respostas ao *quorum* de leitura incluem um *timestamp* e o valor para a chave pedida. Caso exista uma escrita pendente, isto é, uma escrita que já foi replicada porém ainda não foi *commited* pelo *leader*, a resposta deve incluir apenas o *timestamp*.
- A reposta ao pedido de leitura é feita com base no maior *timestamp* recebido, caso se trate de uma escrita pendente é inicializada uma política *back-off* nas próximas tentativas de leitura do *quorum*.
- Foi definido um limite máximo de 5 tentativas, sendo que o intervalo entre cada uma delas é aumentado gradualmente. Este limite foi estabelecido com o objetivo de não sobrecarregar a rede com pedidos no caso de existir alguma situação anômala que não permita realizar a leitura.
- Para minimizar a carga na rede, o *quorum* de leitura não inclui todos os servidores, mas apenas uma amostra aleatória suficiente para formar a maioria. Essa amostra aleatória é alterada a cada tentativa.
- O método responsável por enviar os pedidos para o quorum de leitura (**send_read_quorum**) inicializa um *Timer* para gerar uma tentativa nova ao fim de um intervalo de tempo, a fim de evitar *deadlocks*. Além disso, é usado um identificador único que é incrementado a cada nova tentativa, para garantir que respostas atrasadas não são consideradas.

No artigo é mencionada a possibilidade de dividir aleatoriamente os pedidos entre o *leader* e os restantes servidores, porém, como o protocolo foi utilizado apenas com a ferramenta *maelstrom* e, assumindo que os pedidos são distribuídos igualmente entre todos os servidores, esta probabilidade não foi implementada. O *leader* continua responsável por responder a pedidos de escrita no caso em que estes sejam enviados diretamente para si.

5 Análise de Resultados

5.1 Mudança de líder

A figura 1 mostra um exemplo de uma mudança de líder. Neste caso existe inicialmente uma partição que impede que o servidor n3 conheça o atual *leader* (ou qualquer outro servidor) e que o deixa num ciclo interminável (enquanto que a partição existe) em que tenta ser *leader*, mas não consegue por nunca ter uma maioria de votos. O servidor tenta enviar mensagens do tipo *request_ vote* e inicia o *timer* **election_timeout** mas como não recebe respostas acaba por dar *timeout*, o que faz com que avance bastante no termo.

Assim sendo, quando regressa ao contacto com os restantes servidores e o seu *timer* dá *timeout* o servidor n3 começa então um novo processo de eleição pedindo votos. No entanto, apesar deste servidor ter um termo superior, não está atualizado no que toca às entradas do **log** e, por isso, os restantes servidores não lhe concedem votos. Porém, ao receberem um pedido de voto de um termo superior, os restantes servidores atualizam os seus termos e começam também os seus *timers* para pedirem votos. Eventualmente o servidor n1 torna-se candidato e pede votos. Os restantes servidores votam nele, fazendo dele o novo *leader*.

A imagem apresentada para demonstrar está disponível para consulta com maior qualidade **aqui**.

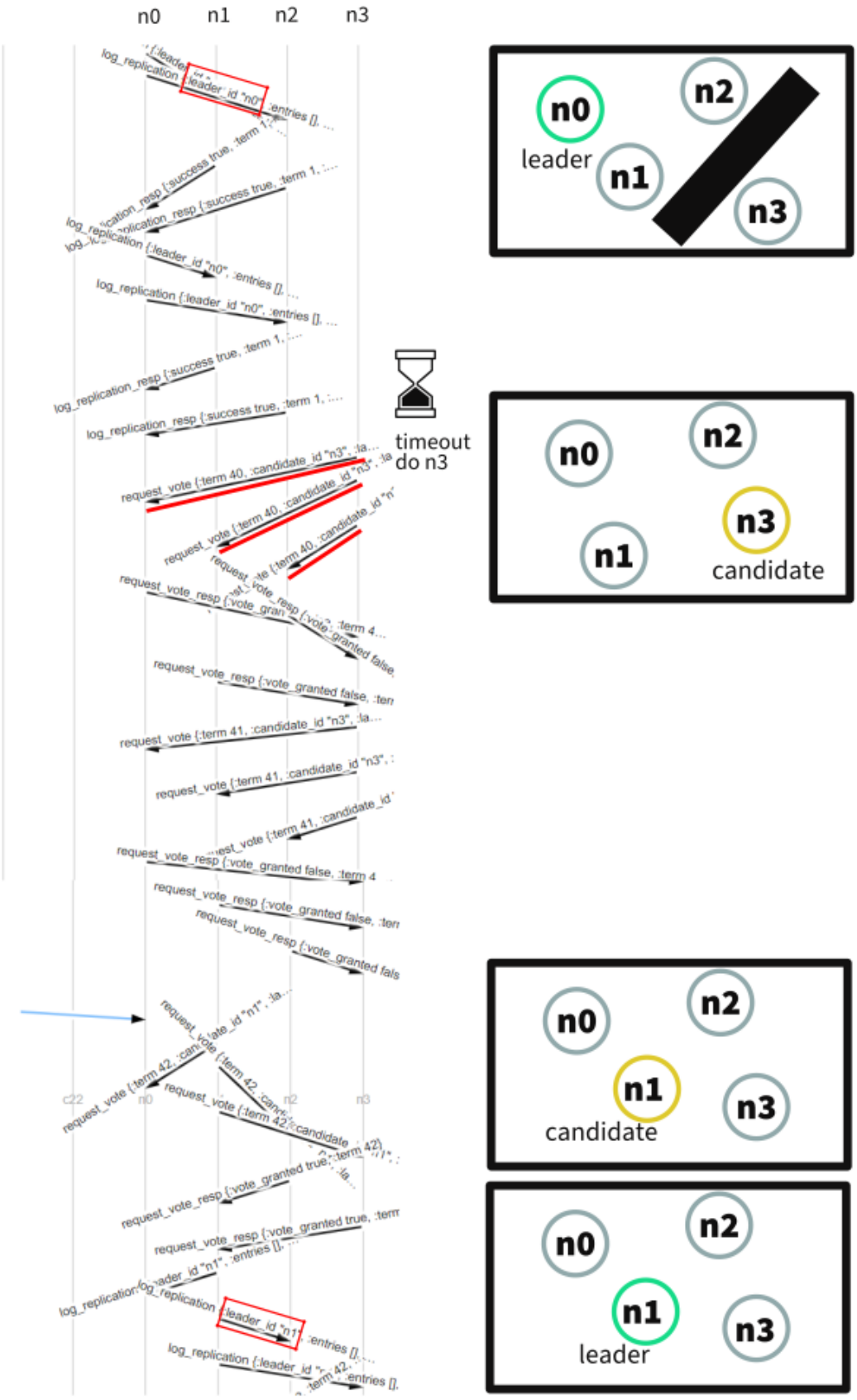

Figura 1: Mudança de líder.

5.2 Leitura direta não linearizável

Como é possível verificar na figura abaixo, as leituras diretas não são linearizáveis. Isto é, se existisse apenas um servidor em que fossem aplicadas as duas operações apresentadas abaixo o resultado não seria o mesmo.

Ao enviar o pedido de escrita (*write{:key 9, :value 3}*), o valor associado à chave 9 passaria a ser 3 e, como tal, ao fazer uma leitura posterior dessa chave (*read{:key 9}*) seria lido o valor 3. No entanto, como se pode constatar pela imagem abaixo, o valor lido é 0.

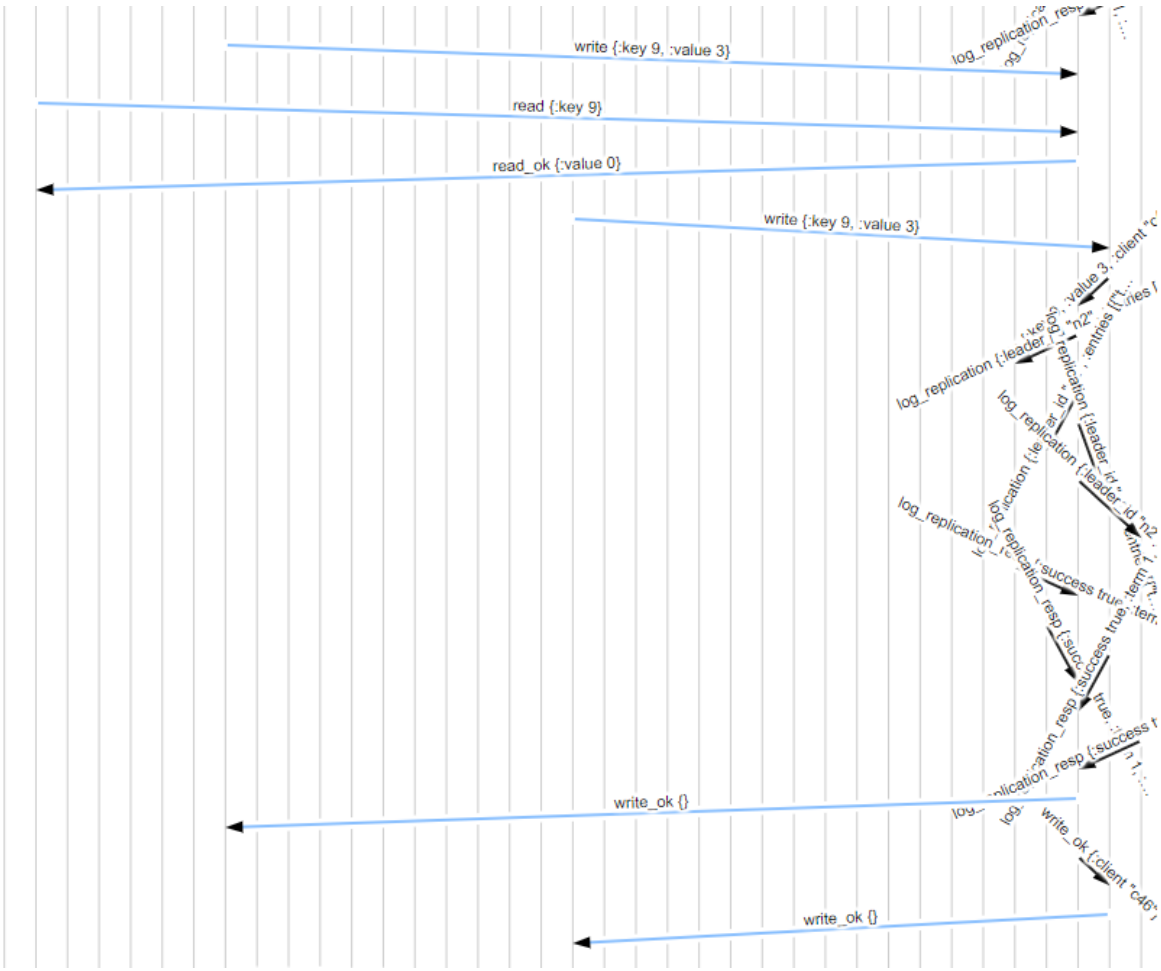

Figura 2: Leitura direta não linearizável.

Isto acontece porque, apesar do pedido de escrita ter chegado primeiro do que o de leitura, é necessário que a entrada no **log** resultante da escrita seja replicada na maioria dos servidores e aplicada no *leader* para que quando o mesmo recebesse o pedido de leitura respondesse com o valor atualizado.

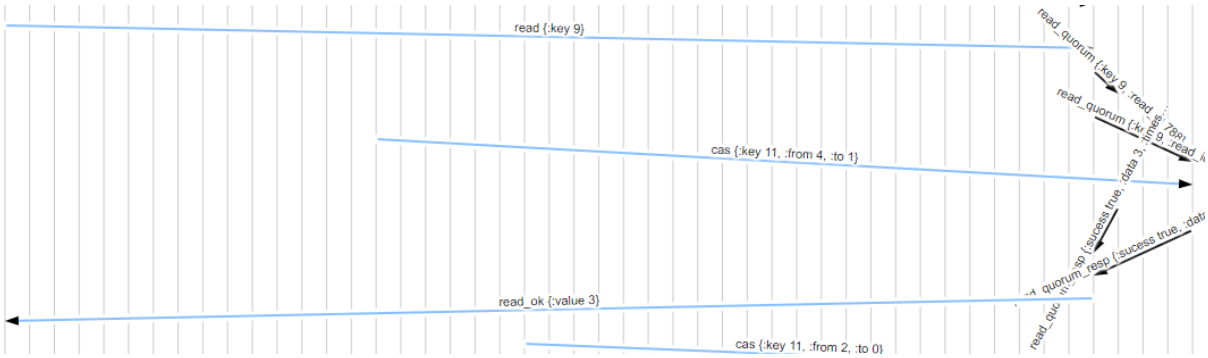

Figura 3: Leitura com o valor atualizado.

5.3 Comparação de desempenho entre Raft e esta variante

A comparação de desempenho entre ambos foi dificultada devido à natureza dos testes realizados pela ferramenta *maelstrom*, isto é, a variante torna-se mais eficiente quando existe uma carga maior no líder e, dessa forma, é possível tirar partido dos restantes servidores. Porém, por alguma razão desconhecida do grupo, o *maelstrom* consegue "perceber"qual é o nodo líder, provavelmente pela latência das respostas, e envia para este a maioria dos pedidos de leitura.

Tendo isto em conta, os resultados apresentados acabam por não ser os mais desejáveis. Porém é possível, mesmo assim, identificar algumas diferenças entre o protocolo **Raft** e a sua variante. Ambos os resultados apresentados nas figuras 4 e 5 foram gerados pelo *maelstrom* para uma rede com 4 servidores e uma frequência de 100 pedidos por segundo, de forma a gerar uma carga de trabalho alta.

- A maioria das respostas *read ok* apresentam latência sensivelmente mais próxima de 1ms na variante quando em comparação com o protocolo Raft.
- Na variante é possível identificar respostas com uma latência acima de 100 ms, correspondendo aos casos em que existem escritas pendentes e é iniciada uma política de *back-off*.
- A latência das respostas *write ok* e *cas ok* são idênticas.

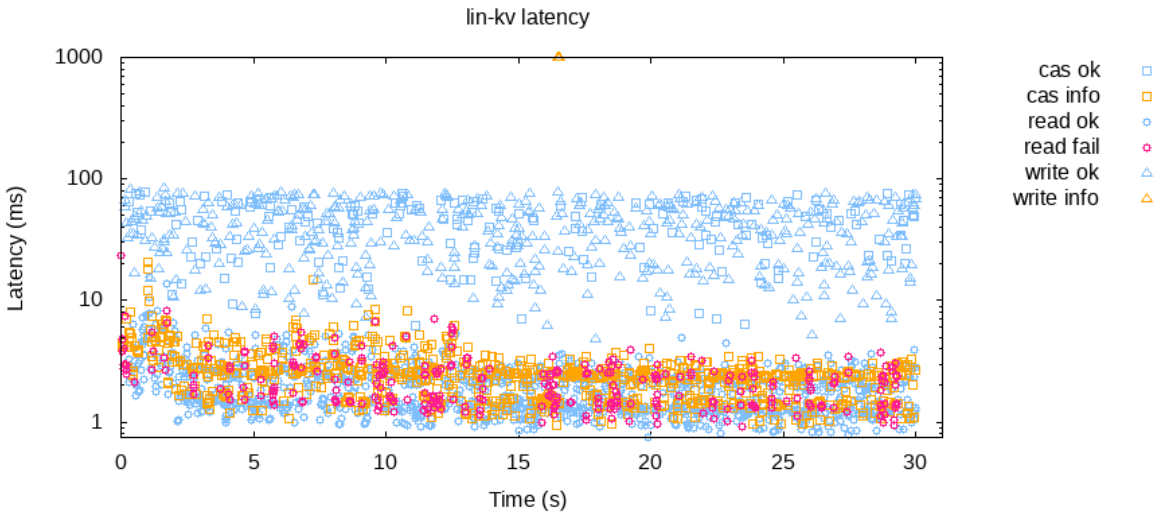

Figura 4: Latência dos pedidos no protoco **Raft**

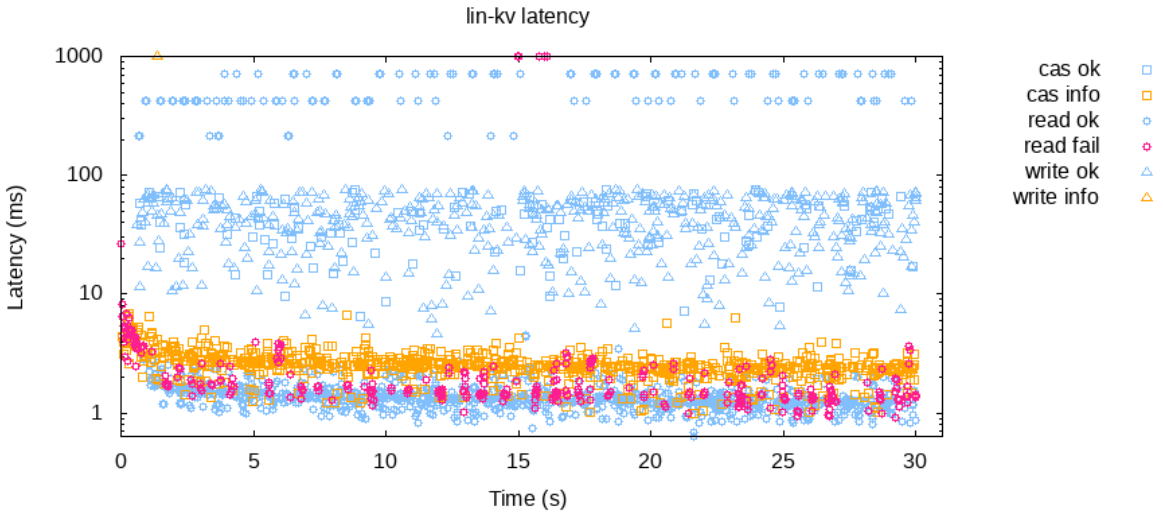

Figura 5: Latência dos pedidos na variante do protoco **Raft**

Em suma, a variante, num ambiente normal, terá sempre resultados piores em termos de latência de pedidos de leitura quando comparada com o protocolo **Raft**, uma vez que consultar um *quorum* de leitura adiciona uma latência inexistente quando em comparação com a consulta de um mapa chave-valor. Pelo contrário, em situações adversas com elevado *stress* de pedidos, a variante terá um desempenho melhor visto que retirará carga do líder. A variante também pode ter outras vantagens, nomeadamente na presença de partições. Por exemplo, numa topologia com 4 nodos em que existe uma partição entre um nodo e o líder (ver figura 6), no protocolo um pedido de leitura direcionado a este nodo (n2) iria sempre retornar erro enquanto que na variante era possível responder consultando os outros nós.

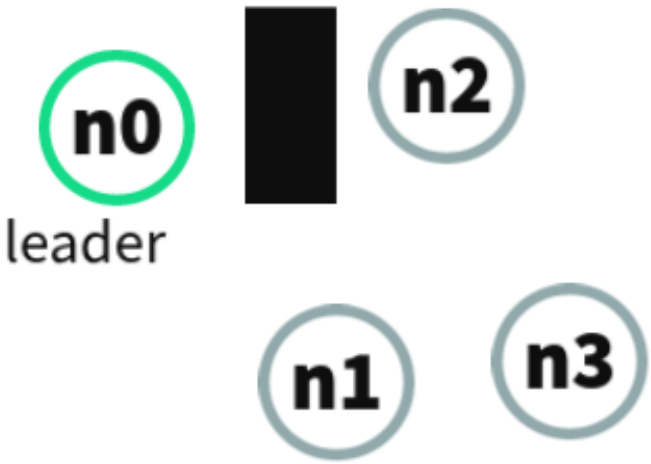

Figura 6: Exemplo de uma partição

6 Conclusão

Este trabalho permitiu consolidar os conhecimentos obtidos nas aulas sobre o funcionamento dos protocolos de consenso mais especificamente do protocolo Raft.

Teria sido interessante explorar mais opções de otimizações que poderiam ser aplicadas nesta versão do protocolo de modo a melhorar a sua eficiência.