Uma extensão de Raft com propagação epidémica

André Gonçalves[0009−0002−1284−094X], Ana Nunes Alonso[0000−0002−0519−9675], José Pereira[0000−0002−3341−9217] e Rui Oliveira[0000−0003−3408−7346]

INESCTEC e Universidade do Minho, Braga, Portugal

Resumo O algoritmo de acordo Raft é reconhecido pela sua facilidade de compreensão e implementação prática, sendo atualmente adotado em sistemas como o Kubernetes. No entanto, tem algumas limitações em termos de escalabilidade e desempenho por concentrar o esforço no líder.Neste trabalho apresentamos um novo algoritmo que expande o Raft com a incorporação de mecanismos de propagação epidémica para descentralizar o esforço da replicação. A nossa proposta é avaliada experimentalmente com uma implementação em Go e testada com um número significativo de processos.

Palavras chave: Acordo distribuído · Raft · Propagação epidémica.

1 Introdução

O acordo é um dos problemas mais fundamentais em sistemas distribuídos tolerantes a faltas. O problema surge da necessidade de processos distribuídos chegarem a um acordo sobre um certo valor proposto, que se torna especialmente complexo na presença de faltas. Bastantes algoritmos foram desenvolvidos, ao longo das últimas décadas, para resolver este problema, cada um com os seus pressupostos e garantias. Entre eles destaca-se o Paxos [6], desenvolvido por Lamport, sob a forma de uma máquina de estados replicada [13], procurando chegar a acordo sobre a ordem em que comandos serão aplicados. No entanto, Paxos é considerado difícil de compreender, o que motivou o desenvolvimento do algoritmo de acordo Raft [10].

Raft garante que um sistema de máquinas de estados funcione como uma única entidade, desde que a maioria dos processos esteja operacional e que mensagens entre estes sejam inevitavelmente entregues. O algoritmo alcança a sua simplicidade ao decompor-se em subproblemas e ao usar liderança forte, onde um processo eleito (líder) assume a responsabilidade de tomar as decisões para todo o sistema. Em operação normal, o líder recebe os pedidos dos clientes, adiciona-os ordenadamente a um registo, que replica pelos outros processos (seguidores). Assim que uma maioria de seguidores informa o líder de que o registo foi replicado com sucesso até um certo ponto, este pode aplicar as entradas até esse ponto e responder aos clientes respetivos. A centralização do algoritmo no líder torna-o simples, mas, em contrapartida, pouco escalável: com o aumento

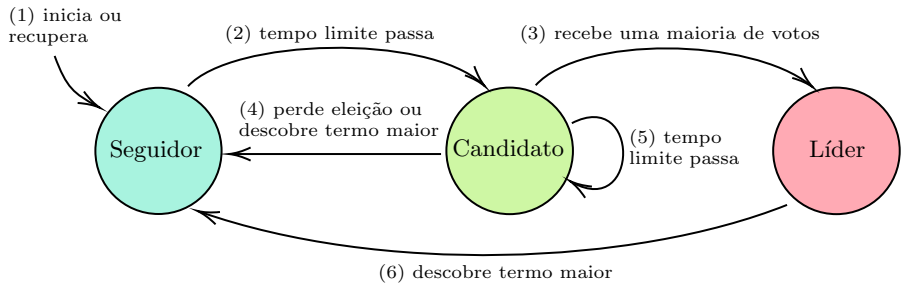


Figura 1: Transições entre estados

do número de réplicas ou carga de trabalho pelos clientes, o líder rapidamente esgota os seus recursos, e torna-se o gargalo do sistema.

Propomos a adição de novos mecanismos baseados em propagação epidémica [8] ao Raft para superar estas limitações. Com propagação epidémica os processos comunicam através do encaminhamento de mensagens entre os seus vizinhos. A nossa abordagem consiste em introduzir no líder rondas de propagação epidémica periódicas, enviadas numa permutação [12], para replicar o registo. Com esta abordagem, as mensagens do líder conseguem alcançar todos os seguidores, mesmo que o líder não esteja diretamente conectado com cada um deles. Desta forma, evitamos que eleições desnecessárias aconteçam, que em Raft são iniciadas por um seguidor quando este não recebe comunicação do líder durante um determinado limite de tempo. Adicionalmente, estudamos a utilização de novas estruturas de dados na propagação epidémica, que permitirão a qualquer processo descobrir o próximo índice do registo até onde as entradas estão confirmadas e poderão ser aplicadas.

Implementamos o algoritmo Raft com e sem os novos mecanismos em Paxi, uma infraestrutura de código aberto para prototipagem e avaliação de algoritmos de replicação escrita em Go [11,1]. As implementações são avaliadas num sistema com 51 processos, cada com um CPU dedicado.

2 Contexto

Raft garante a coerência inevitável de máquinas de estados através da gestão de um registo de operações replicado. Num sistema Raft, os processos assumem, num determinado momento, um de três estados, conforme a Fig. 1. Um processo inicia-se no estado de *seguidor*, que recebe pedidos e responde de acordo, nunca iniciando a comunicação. Este transita para o estado de *candidato* quando assume que o líder falhou, iniciando uma eleição e solicitando ativamente votos aos outros processos para ser eleito. Ao receber uma maioria de votos transita para o estado de *líder*, assumindo toda a responsabilidade de manter o registo replicado coerente e processar os pedidos dos clientes. Referimos-nos a cada processo pelo estado em que se encontra.

No contexto do Raft, o tempo é dividido logicamente em *termos*. Ao termo é associado um número natural que cresce monotonamente, que agindo como um relógio lógico, permite a ordenação total de eventos. O termo dum processo é incrementado quando este inicia (ou reinicia) uma eleição, ou quando aprende de outro processo que o seu termo está desatualizado, ou seja, é menor.

A interação entre processos é efetuada por invocações remotas (*Remote Procedure Calls* / RPCs) assíncronas. Ao iniciar uma invocação remota o processo não bloqueia, permitindo-lhe processar outros pedidos que cheguem entretanto. As invocações são reexecutadas após um tempo limite, para tolerar a perda de mensagens na rede. São necessários dois tipos de RPC: `RequestVote RPC`, iniciados por candidatos para recolher votos numa eleição; e `AppendEntries RPC`, iniciados pelo líder para replicar o seu registo, servindo também como *heartbeat*, para informar os seguidores de que está ativo.

Raft decompõe a sua lógica em três subproblemas: eleição do líder, replicação do registo e garantia de acordo. Uma vez que a nossa proposta se foca apenas na replicação, detalhamos aqui apenas a replicação de registo. Após ser eleito líder, este deve: processar pedidos dos clientes, replicar o seu registo e enviar *hearbeats* para manter a liderança. Os clientes enviam operações para ser aplicadas na máquina de estados replicada. Ao receber um pedido de um cliente, o líder adiciona a operação e termo de receção como uma nova entrada no seu registo. Depois, emite RPCs `AppendEntries` para replicar a nova entrada para cada seguidor. Assim que o líder sabe que a entrada foi replicada com sucesso para uma maioria de processos, ou seja, foi *confirmada*, pode aplicar a respetiva operação na sua máquina de estados e responder com o resultado ao respetivo cliente.

Cada processo tem um valor que cresce monotonamente, `CommitIndex`, que corresponde ao índice no registo da última entrada confirmada. Todas as entradas anteriores ao `CommitIndex` são também consideradas confirmadas. O líder informa via `AppendEntries` do seu `CommitIndex` para que os seguidores possam atualizar os seus próprios. Os processos devem aplicar ordenadamente as novas entradas confirmadas na sua máquina de estados.

O termo de receção e o índice em cada entrada são necessários para detetar incoerência entre registos. O líder incluí nos pedidos `AppendEntries` uma partição do seu registo com as entradas ainda não replicadas com sucesso no respetivo processo destinatário. Ao receber um pedido `AppendEntries`, o seguidor verifica se a entrada anterior às entradas no pedido não entra em conflito com o seu registo e, nesse caso, pode adicionar as novas entradas. Na resposta, o seguidor informa se as entradas foram replicadas com sucesso. No Raft a propriedade de correspondência de registo permite reconhecer se dois registos são iguais até um dado índice. A propriedade diz o seguinte: se dois registos contêm no mesmo índice uma entrada com o mesmo termo, então todos as entradas até esse índice são idênticas. Ao receber a indicação de sucesso na resposta, o líder reconhece que o registo foi replicado com sucesso até ao maior índice das entradas enviadas no respetivo pedido RPC. Caso receba indicação de insucesso, significa que não enviou entradas suficientes no pedido e deve reenviar um novo RPC `AppendEntries` com entradas começando num ponto anterior. O líder po-

derá receber indicação de insucesso várias vezes até chegar a um ponto onde os registos sejam compatíveis.

3 Extensão com Propagação Epidémica

Raft assume conectividade de todos-para-todos entre os processos do sistema desde que a rede não esteja particionada. O líder replica o seu registo via mensagens um-para-um para os seguidores. A nossa ideia consiste em disseminar mensagens para replicação (pedidos `AppendEntries`) no sistema com propagação epidémica, que fornece escalabilidade e robustez mesmo em redes em que a conectividade não é transitiva. Além disso, expandimos ainda mais o algoritmo ao introduzir estruturas de dados no estado dos processos, partilhadas com a propagação epidémica, que permitirá avançar o `CommitIndex` de forma descentralizada.

3.1 Propagação Epidémica de `AppendEntries`

Raft usa RPCs para comunicação entre processos, mas enquanto esta abordagem torna o algoritmo mais simples também o limita. Na nova versão de Raft, o líder dissemina periodicamente o mesmo pedido `AppendEntries` por propagação epidémica e quando um seguidor recebe um pedido pela primeira vez, responde ao respetivo líder. Contudo, pode acontecer que processos não recebam um pedido numa dada ronda de propagação e, portanto, falhar a replicação de entradas. Então, no próximo pedido de `AppendEntries` recebido, este irá responder que a replicação falhou e o líder iniciará a invocação de RPCs `AppendEntries` com o processo até que as entradas em falta sejam replicadas com sucesso.

O líder transmite pedidos `AppendEntries` em rondas de propagação epidémica numa permutação [12] para replicar as entradas que ainda não foram confirmadas. O líder calcula uma permutação de seguidores, isto é uma lista aleatória dos identificadores dos seguidores, que percorre circularmente em cada ronda de propagação epidémica, como especificado no Algoritmo 1. As rondas são iniciadas periodicamente para replicação e para servir de *heartbeat*. O líder inicia periodicamente uma ronda de propagação epidémica de um pedido `AppendEntries`

Algoritmo 1: Ronda de propagação epidémica para o processo $P_i, i \in 0..n-1$

```
Data:
F ← fanout value
c ← 0
u ← permutação de 1 a n exceto i
Function Ronda(m):
    for i ← 0 to F do
        enviar m para u[(c + i) mod F];
    end
    c ← c + F;
```

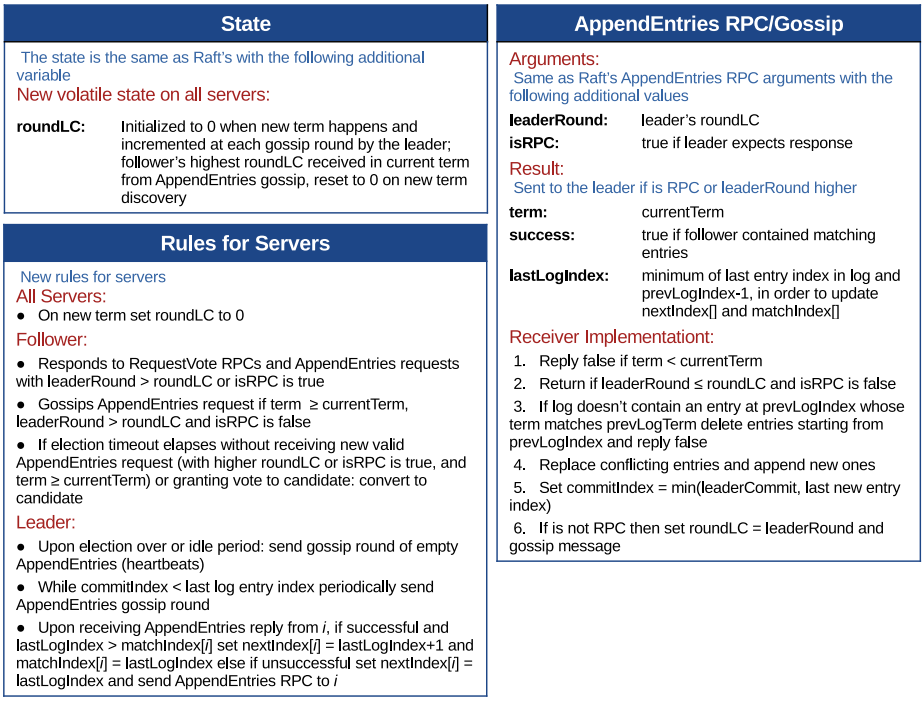


Figura 2: Raft com propagação epidémica de `AppendEntries`.

com as entradas ainda não confirmadas. Por outro lado, se todas as entradas do registo foram confirmadas, o líder pode escolher um intervalo de tempo maior para iniciar rondas com *heartbeat*, para manter a liderança.

Em pedidos `AppendEntries` o líder incluí uma variável booleana que permite aos seguidores distinguir entre pedidos `AppendEntries` provenientes de RPC ou de propagação epidémica. Caso seja um RPC emitido pelo líder, o seguidor tem que responder ao pedido. Se o pedido vem de propagação epidémica então o seguidor responde apenas se for a primeira vez que o recebe.

Para distinguir se os pedidos `AppendEntries` são provenientes de uma ronda de propagação epidémica mais recente, os processos mantêm no seu estado uma variável inteira, `RoundLC`, que serve como um relógio lógico no mandato atual. Cada processo repõe o seu `RoundLC` a zero quando o mandato muda. O líder incrementa o seu `RoundLC` quando começa uma nova ronda e inclui esta variável nos pedidos `AppendEntries` para que os seguidores a possam atualizar no seu estado. Assim, seguidores evitam processar e propagar mensagens antigas ou já recebidas. Seguidores consideram as mensagens mais recentes (com `RoundLC` maior que o seu) um *heartbeat* do líder.

O nosso algoritmo preserva grande parte da lógica do Raft original. A Fig. 2 sumariza os novos mecanismos adicionados e as alterações necessárias para expandir o Raft, mantendo a língua original para facilitar a comparação.

3.2 Novas Estruturas de Dados

A nova versão de Raft diminui o número de mensagens que o líder tem de trocar ao enviar entradas não confirmadas em lote e ao deixar a disseminação destas entradas para os seguidores. Contudo, mantemos a abordagem de que o líder só avança o `CommitIndex` após receber confirmação de replicação de uma maioria de seguidores. Para avançar o valor do `CommitIndex` de forma descentralizada acrescentamos ao Raft novas estruturas de dados para a propagação epidémica. Estas estruturas consistem em três variáveis utilizadas da seguinte forma:

- `Bitmap:` mapa de bits, no qual apenas o processo pode alterar o respetivo bit para "um"; regista a votação para `MaxCommit`
- `MaxCommit:` valor máximo que pode ser atribuído a `CommitIndex`, corresponde ao maior índice confirmado observado pelo processo;
- `NextCommit:` valor a ser considerado para o próximo `MaxCommit`; índice do registo em votação para o próximo valor de `MaxCommit`.

Estas novas variáveis são atualizadas e partilhadas pelos processos nos pedidos `AppendEntries`, garantindo a coerência e convergência dos valores. Os processos usam o mapa de bits para descobrir quando uma maioria replicou corretamente o registo até `NextIndex`. Cada processo deve colocar o seu bit no `Bitmap` a "um" quando o seu registo possui a entrada em `NextIndex` e o mandato da última entrada é igual ao mandato atual. Definimos duas funções para atualizar `NextCommit` e `MaxCommit`: `Update`, para atualizar os valores quando a maioria é alcançada no mapa de bits; e `Merge`, para combinar os valores recebidos de diferentes processos durante a propagação epidémica. O algoritmo garante que `NextCommit` é maior que `MaxCommit` antes e após o uso das funções `Merge` e `Update`.

A função `Update`, conforme o Algoritmo 2, atualiza os valores de `MaxCommit` e `NextCommit` e repõe o `Bitmap` a "zeros" quando este demonstra uma maioria (linha 1). Isto significa que uma maioria dos processos colocou "um" no `Bitmap`, ou

Algoritmo 2: Função para atualizar `Bitmap`, `NextCommit` e `MaxCommit` para o processo $P_i, i \in 0..n-1$

```
   State: bitmap; maxCommit; nextCommit; commitIndex
   Function Update():
1  |  if count of "1"s in bitmap higher or equal than majority size then
2  |  |  maxCommit ← nextCommit;
3  |  |  bitmap ← {0, 0, ..., 0};
4  |  |  if nextCommit ≥ index of last entry or term of last entry is not equal
   |  |  to current term then
5  |  |  |  nextCommit ← nextCommit + 1;
6  |  |  else
7  |  |  |  nextCommit ← index of last entry;
8  |  |  |  bitmap [i] ← 1;
9  |  |  end
10 |  end
```

Algoritmo 3: Função para combinar `bitmap`, `nextCommit` e `maxCommit` recebidos nos pedidos `AppendEntries` para o processo $P_i, i \in 0..n-1$

```
   State: bitmap; maxCommit; nextCommit
   Function Merge(bitmap', maxCommit', nextCommit'):
1  |  maxCommit ← Max(maxCommit, maxCommit');
2  |  if nextCommit ≤ nextCommit' then
3  |  |  bitmap ← BitwiseOR(bitmap, bitmap');
4  |  end
5  |  if nextCommit ≤ maxCommit then
6  |  |  bitmap ← bitmap';
7  |  |  nextCommit ← nextCommit';
8  |  end
```

seja, replicaram o seu registo até `NextCommit` e, então, `NextCommit` e `MaxCommit` podem avançar. `MaxCommit` recebe o valor de `NextCommit` e o `Bitmap` é reposto com "zeros" (linhas 2 e 3). Depois, `NextCommit` é atualizado conforme o estado atual do registo local: se `NextCommit` está num ponto mais avançado que o registo local ou o registo não possuí nenhuma entrada com o mandato atual (linha 4) então o valor de `NextCommit` é incrementado (linha 5); senão, o processo tem uma entrada a seguir ao `NextCommit` atual, com o mandato atual e, portanto, avança-se `NextCommit` para o índice da entrada mais avançada no registo (linha 7). Além disso, satisfazem-se as condições para colocar a "um"o valor respetivo ao processo no `bitmap` (linha 8).

A função `Merge` combina os valores com os recebidos de outros processos nos pedidos `AppendEntries` conforme o Algoritmo 3. Primeiro, `MaxCommit` é atualizado para o maior entre o valor no estado e o recebido (linha 1). Depois, se `NextCommit` local for menor ou igual que o recebido significa que a informação no `Bitmap` recebido pode ser incluída no `Bitmap` local através de um "ou lógico" bit a bit (linhas 2 a 4). Caso uma maioria de processos tenha já replicado o registo até `NextCommit` (linha 5), é necessário substituir `Bitmap` e `NextCommit` com os recebidos (linhas 6 e 7), dado que estão mais avançados.

Os seguidores podem atualizar o seu `CommitIndex` para o mínimo entre o índice da última entrada no registo e o `MaxCommit` se o mandato da última entrada no registo for igual ao mandato atual.

Raft garante a coerência inevitável dos registos replicados, permitindo que o número de entradas nos registos difira entre processos. Ao atualizar `NextCommit` escolhe-se o maior valor possível conforme o estado do registo do processo. Contudo, um novo líder pode ser eleito com um registo menor que este `NextCommit`, o que pode causar atrasos na confirmação de entradas. Para resolver este problema, as variáveis `NextCommit` e `Bitmap` devem ser redefinidas quando uma eleição é iniciada ou se se descobre que foi iniciado um novo mandato. Processos, nestes casos, devem repor o `bitmap` a "zeros" e atribuir o valor de `MaxCommit+1` a `NextCommit`, pois é garantido pelo Raft que um processo só pode ser eleito líder se possuir o seu registo replicado até `MaxCommit`, uma vez que uma maioria de processos já tem o seu registo confirmado até esse ponto.

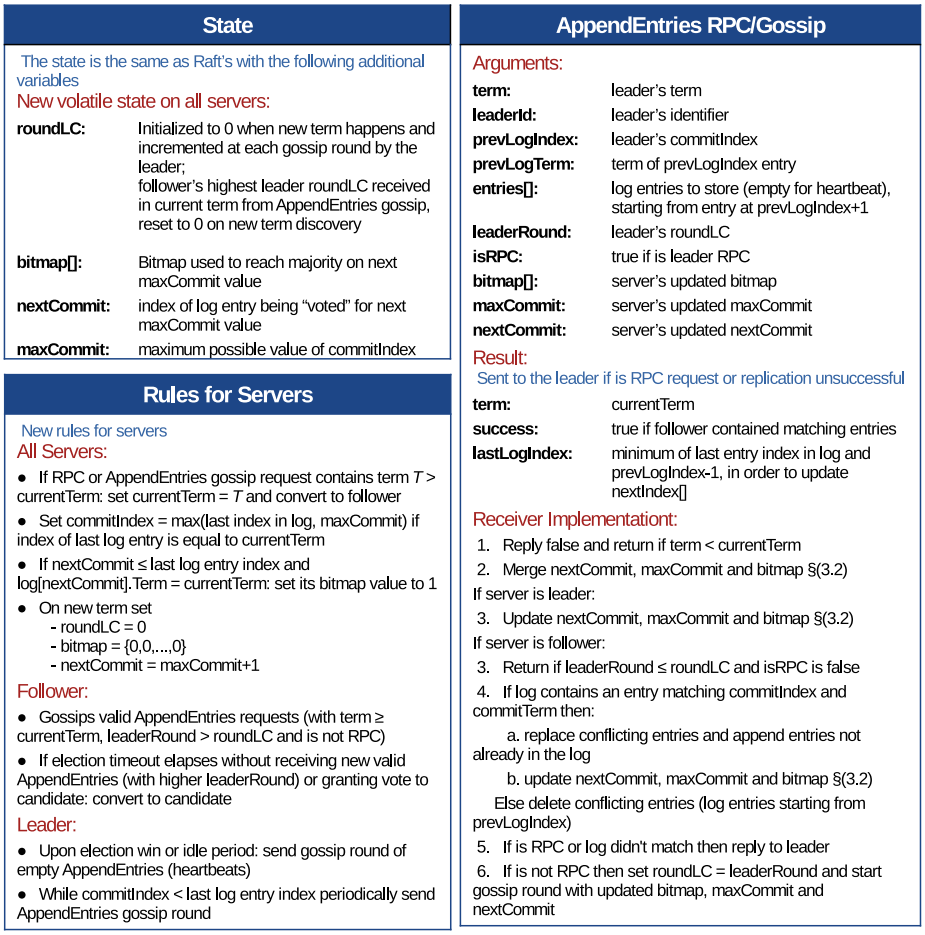


Figura 3: Raft com propagação epidémica de `AppendEntries`

Caso o líder falhe sem ter descoberto que um seguidor encontrou um `MaxCommit` maior que o seu, o próximo processo eleito possuirá todas as entradas confirmadas, ou seja, as entradas do registo do líder até ao maior `MaxCommit` descoberto. Segundo o Raft, um processo pode ser eleito se o seu registo possuí todas as entradas confirmadas pelo líder. Um candidato é eleito se receber uma maioria de votos. Numa eleição cada seguidor vota no primeiro candidato que pede o seu voto e possuí o seu registo tão ou mais avançado com o seu, de acordo com a propriedade de correspondência de registo. O valor do `MaxCommit` indica que uma maioria de processos replicou com sucesso o registro do líder até a este índice. Portanto pela restrição de eleição do Raft apenas candidatos que conseguiram replicar o seu registo até ao maior `MaxCommit` descoberto conseguirá receber uma maioria de votos e tornar-se líder. Assim qualquer novo possível líder possuirá todas as entradas confirmadas em termos anteriores. A Figura 3 sumariza as novas alterações do algoritmo relativamente ao Raft.

4 Avaliação Experimental

Fazemos uma análise comparativa entre as duas novas versões de Raft e o original. Para tal usamos Paxi [11,1], uma infraestrutura para prototipagem e avaliação de algoritmos de replicação escrita em Go.

4.1 Configurações

Corremos as experiências numa única máquina com 128 núcleos, incluindo os processos Raft (réplicas) e os clientes. Cada réplica corre em apenas um núcleo dedicado, para reduzir a variação dos resultados causado pela partilha de recursos. Usamos o cliente Paxi para produzir carga no sistema e realizar a avaliação. O cliente Paxi permite simular vários clientes concorrentes com ou sem uma taxa de pedidos determinada. Cada cliente envia um pedido e espera pela resposta, antes de enviar o próximo. Os testes são executados apenas na fase de replicação do algoritmo com um líder estável.

Comparamos o desempenho, disponibilidade e escalabilidade do Raft original e das duas novas versões. Chamamos Versão 1 ao Raft com propagação epidémica de `AppendEntries` e Versão 2 à que usa também as novas estruturas de dados. Para analisar os algoritmos medimos: a latência média de resposta, taxa de processamento de pedidos, o uso de CPU por cada réplica e a latência entre as entradas recebidas pelo líder e a sua confirmação em cada réplica. Corremos cada experiência 3 vezes e usamos a média dos valores resultantes no desenho dos diagramas.

4.2 Resultados

Para analisar o desempenho corremos várias experiências com diferentes taxas de pedidos nos clientes. Em cada experiência usamos 100 clientes concorrentes e 51 réplicas.

A Fig. 4 mostra os resultados da análise de desempenho dos algoritmos. Observamos que as novas versões apresentam um desempenho significativamente melhor do que o Raft original. A Versão 1 alcança uma taxa de pedidos muito maior com um pequeno aumento na latência. Contudo, a Versão 2 atinge o seu limite com um certo taxa de pedidos, e a latência cresce num ritmo mais elevado,

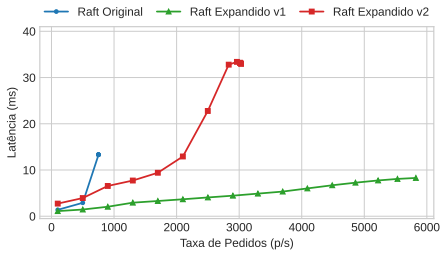
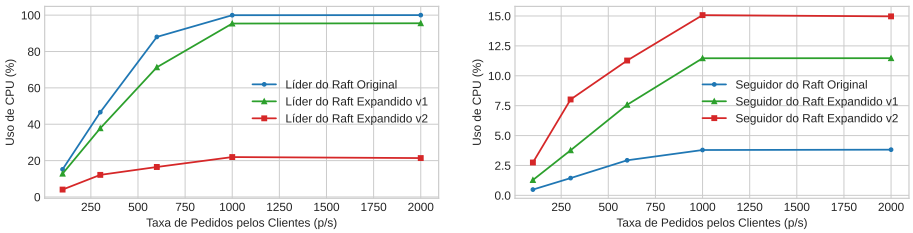
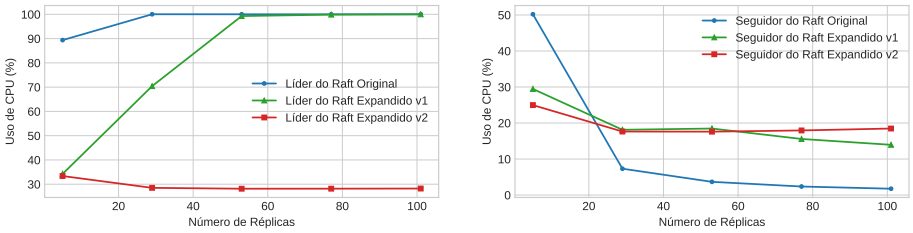


Figura 4: Latência média por taxa de pedidos.



(a) Uso do CPU pelos Líderes (b) Uso do CPU pelos Seguidores

Figura 5: Uso de CPU por taxa de pedidos pelos clientes.



(a) Uso do CPU pelos Líderes (b) Uso do CPU pelos Seguidores

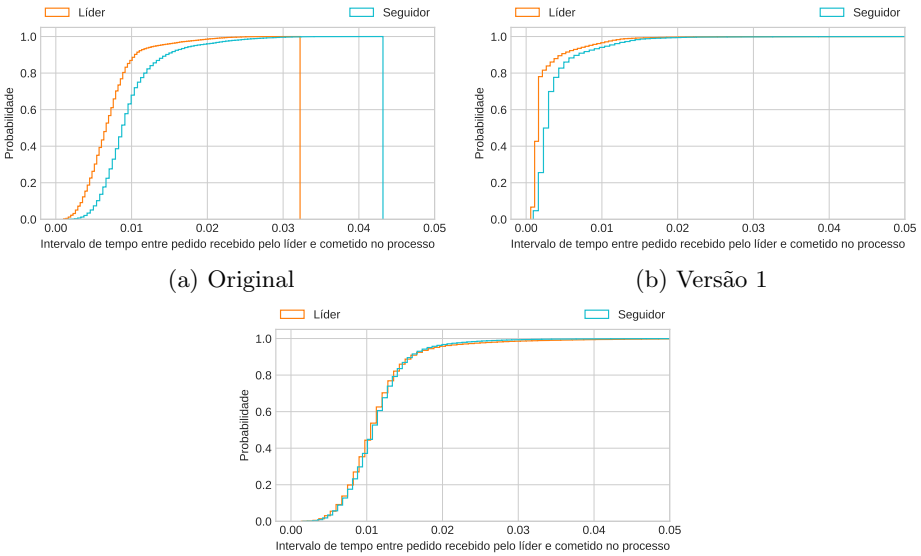
Figura 6: Uso de CPU por número de réplicas no sistema.

devido aos saltos necessários para fazer progredir as estruturas de dados trocadas e consequentemente o `CommitIndex`.

Analisamos também o uso de recursos, nomeadamente do CPU, pelos seguidores e pelo líder com diferentes cargas de trabalho e número de réplicas no sistema. O cliente do Paxi simula 10 clientes concorrentes que enviam pedidos imediatamente após receberem as respostas.

A Fig. 5 mostra o uso de CPU com o aumento da carga de trabalho pelo cliente do Paxi. Observamos que o líder da Versão 1 usou menos recursos que o líder do Raft original, causado pelo uso de propagação epidémica. A Versão 2 mostra um melhor uso dos recursos distribuídos, pois o líder não precisa esperar por respostas diretamente dos seguidores para avançar o `CommitIndex`. Comparativamente aos seguidores, o líder da Versão 2 tem ainda as tarefas adicionais de receber os pedidos dos clientes e iniciar a propagação epidémica, o que resulta ainda assim num uso da CPU ligeiramente superior aos seguidores.

Na Fig. 6 observamos que o Raft original é altamente centralizado no líder, pelo que uso de CPU pelo líder é muito maior que o dos seguidores. O uso de propagação epidémica resultou num melhor uso de recursos distribuídos. A Versão 1 escala melhor com o aumento de réplicas, mas tal como no Raft original o líder torna-se o gargalo do sistema ao chegar a um certo número de réplicas. Na Versão 2, o líder não se torna em nenhum ponto o gargalo do sistema, e o uso de CPU nas réplicas segue uma tendência quase linear, mostrando o melhor uso dos recursos distribuídos entre os três algoritmos.



(a) Original

(b) Versão 1

(c) Versão 2

Figura 7: FDA do intervalo de tempo entre pedido do cliente recebido pelo líder e confirmado na réplica.

Finalmente, medimos a latência entre o momento em que um pedido é confirmado pelo líder e o momento em que é confirmado pelos seguidores. Uma baixa latência aqui é útil pois, caso os clientes estejam dispostos a relaxar as propriedades de coerência, é possível fazer leituras mais rápidas diretamente das réplicas. A Fig. 7 mostra a função de distribuição acumulada (FDA) do tempo que as réplicas demoram a confirmar os pedidos após a receção destes pelo líder. Observamos que os seguidores do Raft original e da Versão 1 demoram mais a confirmar os pedidos, pois o líder é responsável por informar os seguidores do seu `CommitIndex`. Por outro lado, a Versão 2 permite aos seguidores progredirem o `CommitIndex` sem necessitar da confirmação do líder, permitindo que o `CommitIndex` dum seguidor possa estar à frente do líder. Verificamos assim que a Versão 1 consegue ser mais rápida que a original, dada a menor carga. No caso da Versão 2, a decisão descentralizada permite que as réplicas não tenham latência adicional.

5 Trabalho relacionado

Sendo um problema tão importante para sistemas distribuídos tolerantes a faltas, existem inúmeras propostas de algoritmos de consenso, incluindo até alguns que fazem também utilização de propagação epidémica. Concretamente, a primeira proposta nesse sentido [12,7], que introduz a utilização de uma permutação dos destinos de cada mensagem para conciliar a aleatoriedade com o determinismo, baseia-se no algoritmo de Chandra e Toueg [5] (que se destina a tomar apenas

uma decisão e não a ordenar uma sequência) pelo que não é aplicável ao registo do Raft.

Mais recentemente, a propagação epidémica foi utilizada no algoritmo Semantic Gossip [4], de forma semelhante ao Mutable Consensus [12] mas aplicada ao Paxos. Neste caso, não é usada uma permutação dos destinos, pelo que o algoritmo se torna probabilista.

Uma estratégia alternativa para evitar o gargalo na difusão das mensagens pelo líder consiste na pré-difusão dos comandos [2,15] e depois na decisão da sua ordem. Esta abordagem torna no entanto o protocolo mais complicado, pois obriga a relacionar as garantias da pré-difusão com as do consenso e não melhora o processo de recolha de votos.

Os algoritmos de consenso podem também ser descentralizados de forma a eliminar completamente o líder. Neste caso, é acrescentada meta-informação aos comandos sobre os conflitos, de forma a que comandos que não interfiram possam ser ordenados concorrentemente por diferentes processos produzindo assim uma ordem parcial [9]. No entanto, cada processo tem ainda que comunicar diretamente com todos os outros.

Finalmente, a propagação epidémica é também usada nos protocolos de acordo bizantino, normalmente no contexto de *blockchains*, para a propagação das transações [3]. Neste caso, a possível utilização de propagação epidémica para recolha de votos é substancialmente mais complicada [14].

6 Conclusões

A propagação epidémica permitiu descentralizar a replicação do registo do líder, evitando que este se torne o gargalo do sistema, e distribuir de forma mais equilibrada a carga de trabalho entre os processos, resultando num melhor desempenho e uso dos recursos distribuídos. Nomeadamente, a Versão 1 do algoritmo apresentado permitiu aumentar 6× o débito máximo atingível e a Versão 2 diminuir para 1/3 a carga de CPU do líder, ambos em cenários com 51 réplicas.

As novas versões do Raft deverão também ser resilientes em cenários complexos de rede. Como trabalho futuro, será importante avaliar experimentalmente os algoritmos em cenários do mundo real através da *cloud*, com experiências em redes locais (LAN) e redes de larga escala (WAN). Além disso, pretendemos expandir mais os algoritmos para os tornar mais robustos em redes não transitivas. Por exemplo, usar propagação epidémica para recolher votos durante a eleição ou criar mecanismos que permitam seguidores mais atualizados replicar as suas entradas noutros seguidores atrasados que podem ter dificuldades a comunicar com o líder.

Referências

1. Ailijiang, A., Charapko, A., Demirbas, M.: Dissecting the performance of strongly-consistent replication protocols. Proceedings of the 2019 International Conference on Management of Data (2019)

2. Biely, M., Milosevic, Z., Santos, N., Schiper, A.: S-paxos: Offloading the leader for high throughput state machine replication. In: 2012 IEEE 31st Symposium on Reliable Distributed Systems. pp. 111–120 (2012). https://doi.org/10.1109/SRDS.2012.66
3. Buchman, E.: Tendermint: Byzantine fault tolerance in the age of blockchains. Master's thesis, University of Guelph, Canada (2016), http://hdl.handle.net/10214/9769
4. Cason, D., Milosevic, N., Milosevic, Z., Pedone, F.: Gossip consensus. In: Proceedings of the 22nd International Middleware Conference. pp. 198–209. Middleware '21, Association for Computing Machinery, New York, NY, USA (Dec 2021). https://doi.org/10.1145/3464298.3493395, https://doi.org/10.1145/3464298.3493395
5. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (mar 1996). https://doi.org/10.1145/226643.226647, https://doi.org/10.1145/226643.226647
6. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (may 1998). https://doi.org/10.1145/279227.279229, https://doi.org/10.1145/279227.279229
7. Maia, F., Matos, M., Pereira, J., Oliveira, R.: Worldwide consensus. No. 6723 (2011). https://doi.org/10.1007/978-3-642-21387-8_21
8. Montresor, A.: Gossip and epidemic protocols (2017)
9. Moraru, I., Andersen, D.G., Kaminsky, M.: There is more consensus in egalitarian parliaments. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. pp. 358–372. SOSP '13, Association for Computing Machinery, New York, NY, USA (Nov 2013). https://doi.org/10.1145/2517349.2517350, https://doi.org/10.1145/2517349.2517350
10. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference (USENIX ATC 14). pp. 305–319 (2014), https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf
11. Paxi: Paxi framework. GitHub repository ([nd]), https://github.com/ailidani/paxi, accessed: 2023-01-05
12. Pereira, J., Oliveira, R.: The mutable consensus protocol. Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004. pp. 218–227 (2004)
13. Schneider, F.B.: Chapter 7 : Replication management using the state machine approach (1993)
14. Silva, F., Alonso, A., Pereira, J., Oliveira, R.: A comparison of message exchange patterns in BFT protocols - (experience report). In: Intl. Conf. on Distributed Applications and Interoperable Systems (DAIS, with DisCoTec). Lecture Notes in Computer Science, Springer (2020). https://doi.org/10.1007/978-3-030-50323-9_7
15. Tennage, P., Desjardins, A., Kogias, E.K.: Mandator and sporades: Robust Wide-Area consensus with efficient request dissemination (Sep 2022), http://arxiv.org/abs/2209.06152