Replicação de Máquina de Estado Baseada em Prioridade com PRaft

Paulo R. Pinho[1], Luciana de Oliveira Rech[1], Lau Cheuk Lung[1], Miguel Correia[2], Lásaro Jonas Camargos[3]

[1]Departamento de Informática e Estatística, Universidade Fedral de Santa Catarina, Brasil

prdepinho@gmail.com, {luciana.rech,lau.lung}@ufsc.br

[2]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal.

miguel.p.correia@tecnico.ulisboa.pt

[3]Faculdade de Computação, Universidade Federal de Uberlândia, Brasil

lasaro@ufu.br

Abstract. *State machine replication is an approach to create fault tolerant distributed systems. The system remains corret while tolerating faults from some of its replicas. These replicas must execute the same sequence of requests, a problem that is solved by total order algorithsm like Raft. some services may need that requests have different priority levels, such that some be executed before others. In this work, we propose an algorithm, PRaft, that introduces the idea of priority based state machine replication to Raft, modifying it to deal with request priority.*

Resumo. *Replicação de máquina de estado é uma abordagem para criar serviços distribuídos tolerantes a faltas. O sistema se mantém correto enquanto tolera a falta de algumas de suas réplicas. Essas réplicas devem executar a mesma sequência de requisições, problema que é resolvido por algoritmos de ordem total como o Raft. Alguns serviços podem precisar que as requisições tenha diferentes níveis de prioridade, para que umas sejam executadas antes de outras. Neste trabalho propõe-se um algoritmo, PRaft, que introduz a ideia de replicação de máquina de estado baseado em prioridades ao Raft, modificando-o para considerar a prioridades das requisições.*

1. Introdução

Sistemas de computação frequentemente estão sujeitos a faltas, com as quais deve lidar para ser um sistema confiável, evitando que faltas levem a *erros*, e erros levem a *falhas*. Mascaramento é uma abordagem de tolerância a faltas que consiste em adicionar processos redundantes que substituam processos que venham a sofrer faltas.

Replicação de máquina de estado (SMR) é uma técnica bem conhecida de tolerância a faltas em serviços distribuídos [Schneider 1990]. Em SMR um serviço é implementado por um conjunto de *máquinas de estado*, ou *réplicas*. Cada máquina de estado implementa o mesmo serviço, e devem executar a mesma sequência de operações. As operações devem ser determinísticas, de modo que duas máquinas de estado que iniciem no mesmo estado e executem a mesma sequência de operações para que levem o sistema

ao mesmo estado final e cheguem aos mesmos valores de retorno. Mesmo que algumas réplicas se desviem desse comportamento, as suas faltas são mascaradas, isto é, escondidas do cliente.

Para que todos os processos executem a mesma sequência de requisições, elas devem ser ordenadas usando um *protocolo de difusão de ordem total*. No centro desse protocolo mora o *problema do consenso*, em que um grupo de processos deve concordar com um único valor dentre um conjunto de valores propostos. A ordem total das requisições pode ser alcançada encadeando múltiplas instâncias de consenso, cada uma para decidir a ordem de uma requisição [Hadzilacos and Toueg 1993]. Já foi, contudo, demonstrado que não há algoritmo determinístico que resolva o problema de consenso sob a premissa de comunicação assíncrona [Fischer et al. 1985]. Para contornar esse obstáculo algoritmos geralmente estendem o requisito de tempo. Os algoritmos resolvem o problema de consenso garantindo sempre as propriedades de segurança(*safety*), mas garantem as propriedades de terminação apenas em casos específicos, previstos pelo novo requisito de tempo. Uma extensão comum é assumir sincronia mais fraca, como pela existência de um ponto no tempo a partir do qual faltas possam ser detectadas com confiança [Chandra and Toueg 1996]. Propuseram-se muitos algoritmos para resolver esse problema em ambos o modelos de faltas de *crash* [Lamport 2001, Lamport 1998, Oki and Liskov 1988, Ongaro and Ousterhout 2014] e bizantinas [Castro and Liskov 1999, Kotla et al. 2007]. Um algoritmo de consenso é o Raft [Ongaro and Ousterhout 2014], que garante segurança (*safety*) enquanto tolera f faltas em um sistema com $2f + 1$ processos.

Sistemas confiáveis podem ter requerimentos de tempo real, isto é, precisam que operações sejam executadas antes de prazos rígidos [Verissimo and Rodrigues 2001]. Atender tal requsito requer o uso de infraestruturas especializadas que permitam previsibilidade de comunicação e de execução. Isso é dificilmente o caso em sistemas distribuídos de uso geral, nos quais a comunicação inconfiável pode causar atrasos ou perdas de mensagens. Contudo é possível nesse caso atingir requerimentos menos estritos de tempo real – *soft* – [Locke 1986].

Muitos serviços podem tirar proveito de garantias de tempo real *soft* expressas em termos da diferença de níveis de urgência, ou prioridade. Por exemplo, diferentes prioridades podem ser atribuídas a usuários com contratos de serviço distintos em serviços de armazenamento na nuvem [Buyya et al. 2008, Bessani et al. 2011]. Alguns exemplos de serviços que se tornaram confiáveis usando SMR e que poderiam tirar proveito de prioridades são sistema de arquivos em rede [Castro and Liskov 1999, Kotla et al. 2007], serviços de *backup* cooperativo [Aiyer et al. 2005], e sistemas de gerência de bancos de dados relacionais [Luiz et al. 2011].

Há alguns trabalhos sobre algoritmos de difusão de ordem total baseada em prioridades [Nakamura and Takizawa 1992, Rodrigues et al. 1995, Wang et al. 2002]. O assunto apareceu primeiro em [Nakamura and Takizawa 1992], mas o algoritmo apresentado é sobre o modelo síncrono, e não é tolerante a faltas. Os outros trabalhos consideram o modelo de tempo assíncrono, mas precisam de algum *framework* baseado em detectores de faltas. O algoritmo de [Rodrigues et al. 1995] usa *view atomic multicast* [Schiper and Ricciardi 1993], e o algoritmo de [Wang et al. 2002] conta com o *general agreement framework* [Hurfin et al. 1999]. Essas abordagens resultam em um considerá-

vel número de mensagens trocadas entre as réplicas, e como resultado, têm complexidade de mensagens de $O(n^3)$ e $O(n^2)$ respectivamente.

Este artigo trata da extensão de SMR com prioridades, para que requisições de mais alta prioridade sejam executadas antes que as outras. Este problema é chamado de *replicação de máquina de estado baseado em prioridade* –PB-SMR. O problema SMR regular pode ser resolvido empregando um protocolo de ordem total como Raft para entregar a mesma sequência de requisições a todos os processos. Resta fazer com que a sequência seja ordenada segundo a prioridade das requisições.

É proposta neste artigo a modificação do algoritmo Raft [Ongaro and Ousterhout 2014] para lidar com a prioridade das requisições, sem qualquer premissa mais forte do que *eventual synchrony* [Dwork et al. 1988]. O protocolo resultante chama-se PRaft e resolve o PB-SMR dentro da complexidade linear de mensagens.

2. Trabalhos Relacionados

Não há muitos trabalhos de consenso baseado em prioridade na literatura. O pioneiro parece ser [Nakamura and Takizawa 1992], que propõe dois algoritmos. Um deles (PriTO) é uma difusão de ordem total baseado em prioridade no qual mensagens são entregues segundo as suas prioridades. Esse algoritmo depende, contudo, de um modelo de tempo síncrono, e não é tolerante a faltas. Diferentemente, a abordagem deste trabalho considera o modelo de tempo *eventual synchronous*, onde há um tempo de estabilização global depois do qual o sistema se comporta de maneira síncrona.

O artigo de [Rodrigues et al. 1995] propõe um algoritmo tolerante a faltas. Nesse protocolo, requisições de baixa prioridade são adicionadas no final da sequência de cada réplica, assumindo que as requisições são ordenadas a partir do início, onde ficam as de alta prioridade, para baixo, onde ficam as de baixa prioridade. Requisições de alta prioridade são inseridas num ponto no meio da sequência, baseado no progresso da réplica mais rápida. Antes de executar a nova requisição de alta prioridade, todas as réplicas precisam executar todas as requisições de mais baixa prioridade que a réplica mais rápida já executou. Por isso, o algoritmo não interrompe requisições de baixa prioridade para executar um requisição pronta de alta prioridade. O algoritmo assume a existência de um serviço de comunicação de grupo *virtual synchrony* que toma conta da transmissão de mensagens, gerenciamento de visão, tolerância a faltas e difusão atômica [Schiper and Ricciardi 1993]. A falta da capacidade de interrupção resulta em requisições de alta prioridade terem que esperar a execução de requisições de baixa prioridade antes de serem executadas. Essa é a principal diferença com este trabalho, já que o PRaft evita a inversão de prioridade interrompendo e reordenando requisições não confirmadas.

O trabalho de [Wang et al. 2002] propõe um algoritmo tolerante a faltas depende de *general agreement framework* de [Hurfin et al. 1999], que usa um detector de falta $\diamond S$ [Chandra and Toueg 1996] para resolver consenso. Cada vez que o sistema recebe uma requisição r, o serviço de consenso é invocado para definir o número de sequência da nova requisição. Requisições que são de prioridade mais baixa que r são movidas para a posição seguinte, para que r ocupe a sua posição na sequência. Se a réplica já havia começado a executar quaisquer requisições entre as que haviam sido movidas, então a réplica interrompe execução e restaura o estado de antes do processamento dessas requisições.

Quando a maioria das réplicas termina o processamento de r, r está confirmado e numa posição irreversível, donde outra requisição de prioridade mais alta não pode tirá-lo.

A principal diferença entre [Wang et al. 2002] é PRaft, e a principal motivação deste trabalho, é não usar *generic agreement framework* que faz as réplicas trocar mais mensagens que em PRaft. Em [Wang et al. 2002], dada uma lista de requisições para executar, o *framework* do algoritmo de consenso é invocado uma vez e mais $f + 1$ vezes, pelo menos, para cada requisição que confirma. Em PRaft, o acordo é invocado apenas uma vez para cada requisição, e mais uma vez quando ela é confirmada. Isso é alcançado lidando com a prioridade das requisições junto com o protocolo de consenso.

3. Modelo de Problema

O sistema é composto por um conjunto de Π processos, dos quais $f < |\Pi|/2$ podem faltar. Cada processo p mantém uma sequência *log* de requisições, um *termo* e um índice da requisição em processamento e. Uma requisição r no índice i do *log* do processo p é representado por $log_p[i]$.

Prioridade é denotada $P(r)$, e quando uma requisição r tem prioridade maior que r', então $P(r) > P(r')$.

Assume-se que um modelo de faltas em que os processos que faltam não se recuperam, chamado de faltas de *crash*.

Comunicação entre réplicas é feito apenas com troca de mensagens. Mensagens podem se perder, são entregues no máximo uma vez e não estão sujeitas a corrupção.

Considera-se o modelo assíncrono ampliado com o tempo global de estabilização (GST), chamado *eventual synchrony* [Cristian and Fetzer 1998, Fetzer and Cristian 1995]. O sistema comporta-se de maneira assíncrona, em que mensagens podem levar tempo arbitrário de transmissão até a entrega. O sistema mais cedo ou mais tarde entra em um período estável em que as mensagens são transmitidas e as requisições são executadas com sincronia. Esse modelo é necessário para garantir que o algoritmo termine. Períodos de estabilização duram suficiente tempo para o algoritmo terminar, o que acontece com frequência em sistemas reais.

Assume-se que o tempo de execução das requisições seja significativamente longo em comparação com o tempo de execução do protocolo. Se requisições de baixa prioridade são longas, esperar pelo seu término pode ser contra-produtivo para o sistema quando requisições de alta prioridade que demandam atenção imediata se mantêm atrasadas.

Para definir o problema PB-SMR, deve-se definir Inversão Local de Prioridade (LPI) e Inversão Global de Prioridade (GPI), segundo [Wang et al. 2002]. Uma requisição r está invertida no processo p ($LPI_p(r)$) se r está preparado para execução p enquanto executa outra requisição de menor prioridade que r. Uma requisição r está invertida globalmente ($GPI(r)$) quando, para cada processo p que terminou a execução de r, $LPI_p(r)$. Para que não ocorra $GPI(r)$, é necessário que pelo menos um processo tenha executado r sem inversão.

O algoritmo PRaft satisfaz as seguintes propriedades:

- Não trivialidade: Se uma requisição r foi confirmada por qualquer processo, então r foi proposto.

- Acordo: Se quaisquer dois processos p e q confirmaram $log[i]$, então $log_p[i] = log_q[i]$.
- Ordem de prioridade: Requisições não confirmadas são confirmadas segundo a sua prioridade.

4. Raft

Raft é um algoritmo de consenso desenvolvido para ser simples de fácil entendimento. Paxos [Lamport 1998], apesar de ser um algoritmo de consenso mais conhecido, é notável pela sua complexidade. Compreendê-lo é um desafio para muitas pessoas, mesmo pesquisadores experientes [Ongaro and Ousterhout 2014].

Raft tem como objetivo fazer com que os processos concordem em executar a mesma sequência de requisições, enquanto toleram o *crash* de uma parcela de processos. A relação entre o total de processos n e o total de faltas f é dada pela equação: $n = 2f+1$. Uma requisição está confirmada quando a maioria dos processos concordou em executá-la. Em cada processo há uma máquina de estado ao qual ele aplica requisições para processamento. Para uma requisição ser aplicada à máquina de estado ela precisa estar confirmada e todas as requisições anteriores devem estar processadas. Um processo envia a mensagem de retorno de uma requisição ao cliente quando a máquina de estado termina o seu processamento.

Raft é constituído de um conjunto de processos, cada um dos quais está em um dos estados *líder*, *seguidor* ou *candidato*. Normalmente um dos processos é líder, e os demais os seus seguidores. As mensagens vão sempre do líder para os seguidores e dos seguidores respondendo ao líder, exceto na eleição, como logo será explicado. Há dois tipos de mensagens trocadas pelos processos: *AppendEntry*, usado pelo líder para replicar o seu *log*, e *RequestVote*, usado pelo candidato de uma eleição para pedir votos.

O tempo é dividido em *termos*. Um termo é o período que começa com a eleição de um candidato e termina com a falta do líder eleito. É representado por um número, e todos os termos sucessivos são maiores que os anteriores. O termo atual é sempre enviado junto a qualquer mensagem transmitida no sistema. Quando um processo recebe uma mensagem com o termo menor que o seu, ele a ignora. Quando o processo recebe uma mensagem de termo maior, então ele atualiza o seu termo para ficar igual e torna-se seguidor do líder vigente. No máximo apenas um líder opera em cada termo.

A Figura 1 ilustra a sequência de passos do algoritmo desde a eleição até a confirmação de requisições. Essa sequência é composta por três passos. O primeiro passo é a eleição do líder (*RequestVote* e *Vote*), que acontece uma vez no início de cada termo. Eleito o líder, ele passa a receber requisições de clientes replicando-as no segundo passo (*AppendEntry* e Resposta). O terceiro passo é a confirmação das requisições aos seguidores (*AppendEntry*), mensagem que pode coincidir com a replicação de requisições que chegaram depois do *AppendEntry* anterior. Esses passos serão explicados a seguir.

Todos os processos iniciam no estado *seguidor*, que espera receber periodicamente mensagens *AppendEntry* do líder para se manter no seu mandato. Se o seguidor não recebe *AppendEntry* dentro de um *timeout*, ele assume que o líder falhou e inicia uma eleição candidatando-se para ser o próximo líder.

O processo muda o seu estado para *candidato*, incrementa o seu termo e envia

uma mensagem *RequestVote* com o termo atual e o índice da última entrada do seu *log*. Um processo que recebe essa mensagem responde se o seu termo for menor que o do candidato e o seu *log* não for mais atualizado que o dele. Um log_p é mais atualizado que um log_q se o termo da última entrada de log_p for maior que o termo da última entrada de log_q, ou, se forem iguais, se o índice da sua última entrada for maior.

Um candidato que recebe o voto da maioria dos processos muda o seu estado para *líder*, passa a receber requisições de clientes e a enviar mensagens *AppendEntry* periodicamente para replicar requisições e manter o seu mandato.

Se dois ou mais processos se candidataram mais ou menos ao mesmo tempo e os votos ficaram divididos entre eles, pode acontecer de nenhum deles ganhar a maioria dos votos. Se um candidato não for eleito dentro de um *timeout*, ele incrementa o seu termo e elege-se de novo. Esses *timeouts* são aleatórios dentro de uma faixa de milissegundos, para impedir que múltiplos processos candidatem-se sempre simultaneamente.

O *log* de um processo é uma lista de entradas na forma (r, t), onde r é a requisição, e t é o termo em que a entrada foi incluída no log. O líder, ao receber requisições de clientes, anexa as requisições no final do log.

A mensagem *AppendEntry* que o líder envia periodicamente não serve só de *keep-alive* para os seus seguidores, mas também para replicar as requisições que o líder acumula no seu *log* entre um *AppendEntry* e outro, enviando a cada seguidor a parte do *log* que lhe falta. Quando o seguidor s responde ao líder informando que o *log* foi replicado com sucesso, o líder atualiza o $match\_index_s$, que indica até onde o *log* dos dois estão iguais. Quando o $match\_index$ de f seguidores tiver atingido um índice i, então atualiza-se o $commit\_index = i$. Até o índice i as requisições estão confirmadas e o líder pode aplicá-las na máquina de estado e enviar as respostas da execução ao cliente requisitante.

Para o seguidor s replicar as requisições com sucesso, o seu log_s deve estar igual ao do líder até o índice da próxima requisição $next\_index_s$. O seguidor compara o $next\_index_s$ enviado pelo líder com o termo da entrada no índice $next\_index_s - 1$. Se forem iguais, significa que os $logs$ dos dois são iguais até aquele ponto. Se forem diferentes, o seguidor responde ao líder que a replicação foi mal sucedida. Então o líder decrementa $next\_index_s$ e tenta de novo no próximo *AppendEntry*. Cada vez que $next\_index_s$ diminui, maior é a sublista de requisições enviadas ao seguidor no *AppendEntry*, e isso se repete até chegar o ponto em que o log dos dois são iguais e a replicação é bem sucedida.

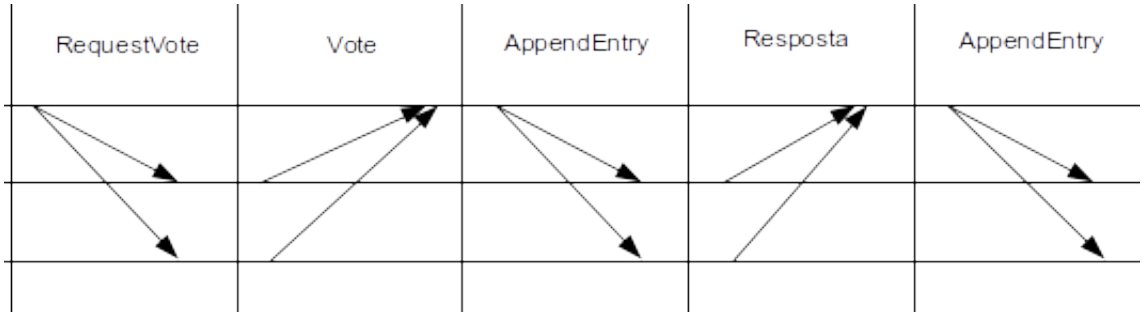


Figura 1. Sequência de passos do Raft

O algoritmo garante que o líder sempre terá todas as requisições confirmadas antes da sua eleição, já que pelo menos um processo com o *log* inteiro sempre está disponível, e o líder eleito é sempre o que tem o *log* mais atualizado.

5. PRaft

As principais contribuições do PRaft (Priority Raft) são a execução antecipada das requisições, a ordem de prioridade e a interrupção de execução. No PRaft os processos não esperam a requisição estar confirmada para executá-la, mas executam-nas na hora em que as recebem. A resposta ao *AppendEntry* não é usado para confirmar as entradas, mas apenas para informar que o *log* foi replicado. A variável $match\_index$ é trocada pela $execution\_index$, pois o critério para a confirmação das entradas não é mais a correspondência entre os *logs* dos seguidores com o do líder, mas com as requisições que os processos tenham terminado de executar.

Os processos mantêm o progresso por meio de dois índices. O progresso de execução é indicado pelo índice e, que marca qual requisição a máquina de estado está executando no momento. E o progresso de confirmação é indicado pelo $commit\_index$, que é incrementado toda vez que a maioria dos processos termina a execução de uma requisição.

Cada requisição r tem uma prioridade definida pelo cliente denominada $P(r)$. O líder sempre ordena as requisições não confirmadas segundo a prioridade delas.

Os valores iniciais das variáveis são a presentadas no Algoritmo 1

O Algoritmo 2 mostra quando o líder l recebe uma requisição r de um cliente c. Calcula-se o valor da posição i que r deve assumir no *log*. Todas as requisições não confirmadas que estiverem em posição maior ou igual a i devem dar espaço para r. Ocorre uma *inversão de prioridade* (linha 2) quando uma dessas requisições é a que está em execução na máquina de estado. Quando isso acontece, a máquina de estado interrompe a execução e restaura o estado até o ponto anterior à posição i. O líder inclui r no *log*, que assume o índice i, e move ao índice seguinte os processos de prioridade menor que r. Então aplica $log[i]$ à máquina de estado, que volta à execução habitual.

Algorithm 1 Valores iniciais

1: $commit\_index \leftarrow -1$;
2: $e \leftarrow \infty$;
3: $next\_index_s \leftarrow 0$;
4: $execution\_index_p \leftarrow -1$.

O Algoritmo 3 descreve o método para definir a posição i da nova requisição r. Novas requisições nunca ocupam posições de requisições confirmadas. Por isso a posição a ser ocupada jamais será menor ou igual a $commit\_index$. A posição é definida levando em consideração a prioridade de r e das requisições não confirmadas do *log*.

Se a requisição r não causou inversão de prioridade, e como consequência a máquina de estado não interrompeu a execução e não restaurou o estado anterior a i, então a máquina de estado segue a execução de $log[e]$ normalmente.

Como consequência da reordenação do *log*, o $next\_index$ de todos os seguidores (linha 11) e $match\_index$ de todos os processos (linha 16) devem ser atualizados para

Algorithm 2 Líder l recebe requisição r de cliente.

1: $i \leftarrow$ posição em que r deve ser inserido no log
2: **if** $e \geq i$ **then**
3: State machine interrompe execução.
4: State machine restaura o estado $snapshot[i-1]$
5: **end if**
6: Insere r no log
7: **if** State machine estiver inativa **then**
8: $e \leftarrow i$
9: Aplica $log[e]$ à State machine
10: **end if**
11: **for** Para cada seguidor s **do**
12: **if** $next\_index_s > i$ **then**
13: $next\_index_s \leftarrow i$
14: **end if**
15: **end for**
16: **for** Para cada processo p **do**
17: **if** $executed\_index_p \geq i$ **then**
18: $executed\_index_p \leftarrow i - 1$
19: **end if**
20: **end for**

Algorithm 3 Determina a posição em que a requisição r é inserida.

1: **for** $i \leftarrow len(log) - 1; i > commit\_index; i \leftarrow i - 1$ **do**
2: **if** $P(log[i]) \geq P(r)$ **then**
3: **return** $i + 1$
4: **end if**
5: **end for**
6: **return** $commit\_index + 1$

que a seção modificada do *log* seja replicada para todos os seguidores e a execução de alguma requisição dessa seção seja desconsiderada.

O Algoritmo 4 mostra os parâmetros do *AppendEntry* que o líder envia periodicamente aos seus seguidores. A variável $entries$ é a sub-lista das entradas que faltam à réplica, todas as entradas do $next\_index_p$ em diante. O seguidor não deve aceitar as novas entradas se o seu *log* não estiver igual ao do líder até o ponto do $next\_index$. São enviados os parâmetros $prev\_index$ e $prev\_term$ para o seguidor fazer a comparação. São enviados também o termo, que vai em todas as mensagens do protocolo, e o $commit\_index$ para informar o seguidor quais requisições estão confirmadas.

O Algoritmo 5 descreve o seguidor quando recebe o *AppendEntry*. O seguidor aceita o *AppendEntry* se seu *log* for igual ao do líder até aquele ponto (linha 1). Como no Raft, o *log* do seguidor deve se conformar com o do líder em toda a seção de índice $prev\_index+1$ em diante. Ocorre inversão de prioridade se o seguidor estiver executando alguma requisição dessa secção (linha 2). Nesse caso, o seguidor faz a mesma coisa que o líder teria feito numa inversão de prioridade. Então o seguidor atualiza o seu *log* incluindo

Algorithm 4 Líder l envia AppendEntry aos seguidores

1: **for** para cada seguidor s **do**
2: $entries \leftarrow log[next\_index_s..]$
3: $prev\_index \leftarrow next\_index_s - 1$
4: $prev\_term \leftarrow log[prev\_index].term$
5: Envia um $AppendEntry(term, entries, commit\_index, prev\_index, prev\_term)$
6: **end for**

Algorithm 5 Seguidor s recebe $AppendEntry\ AE$ do líder l

1: **if** $log_s[prev\_index_{AE}].term = prev\_term_{AE} \vee prev\_index_{AE} = -1$ **then**
2: **if** $e \geq prev\_index_{AE} + 1$ **then**
3: State machine interrompe execução.
4: State machine restaura o estado $snapshot[prev\_index_{AE}]$
5: **end if**
6: Atualiza log
7: Atualiza commit index
8: **if** State machine estiver inactiva **then**
9: $e \leftarrow prev\_index_{AE} + 1$
10: Aplica $log[e]$ à State machine
11: **end if**
12: Responde ao líder "Bem sucedido"
13: **else**
14: Responde ao líder "Mal sucedido"
15: **end if**

todas as requisições que vieram no *AppendEntry*, substituindo as requisições que estavam nesses índices pelas novas. O seguidor então envia uma resposta ao líder indicando que a replicação foi bem sucedida (linha 12).

Se o *log* do seguidor está diferente do *log* do líder até o ponto $prev\_index$, então o seguidor responde com uma mensagem de fracasso (linha 14).

O Algoritmo 6 descreve quando o líder recebe uma resposta do seguidor s. Se a mensagem de operação for bem sucedida, $next\_index_s$ é atualizado (linha 2) para que o líder não envie as mesma requisições a s nos próximos *AppendEntry*. Se a mensagem for de operação mal sucedida (linha 4), então o líder decrementa $next\_index_s$ para enviar uma sublista maior no próximo *AppendEntry*. À medida que o seguidor continua respondendo que a operação foi mal sucedida, o índice vai decrementando e a sublista enviada ao seguidor vai aumentando até achar o ponto em que o *log* dele esteja igual ao do líder, ou ao início, quando o *log* inteiro é replicado.

Os Algoritmos 7 e 8 descrevem o que acontece quando uma máquina de estado termina a execução de uma requisição. Em qualquer processo, seja líder ou seguidor, a máquina de estado está sempre executando requisições, que são aplicadas uma a uma, na ordem em que estão disposta no *log*. Quando a máquina de estado termina a execução de uma requisição, o processo aplica a requisição seguinte na máquina de estado e, se for seguidor, envia uma mensagem ao líder. No Algoritmo 7 o seguidor envia nessa mensagem a variável $exec$, isto é, o índice da requisição que ele acabara de executar

Algorithm 6 Líder l recebe resposta res do seguidor s para o AppendEntry

1: **if** a resposta for "Bem sucedido" **then**
2: $next\_index_s \leftarrow len(log)$
3: **else**
4: $next\_index_s \leftarrow next\_index_s - 1$
5: **end if**

(linha 7). O líder, tendo terminado o processamento ou tendo recebido uma mensagem de término de um seguidor, atualiza $execution\_index_s$ e incrementa o $commit\_index$ se possível(linha 3).

Algorithm 7 State machine de um processo termina a execução da requisição $log[e]$

1: $e \leftarrow e + 1$
2: State machine aplica $log[e]$
3: **if** é o líder **then**
4: $execution\_index_l \leftarrow e - 1$
5: Atualiza o $commit\_index$
6: **else**
7: Envia ao líder a mensagem de fim de execução da requisição em $exec \leftarrow e - 1$.
8: **end if**

Algorithm 8 O líder recebe do seguidor s a mensagem msg de fim de execução

1: **if** $exec_{msg} < next\_index_s$ **then**
2: $execution\_index_s \leftarrow exec_{msg}$
3: Atualiza o $commit\_index$
4: **end if**

O Algoritmo 9 descreve como $commit\_index$ é incrementado. Quando a maioria dos processos tiver executado até o índice i, o líder terá $executed\_index_s \geq i$ para cada processo s que faz parte da maioria dos processos. O $commit\_index$ é o menor valor dentre os $f + 1$ maiores $executed\_index$. Assim todas as requisições que foram executadas pela maioria dos processos são confirmadas.

Algorithm 9 Atualiza $commit\_index$

1: $buf \leftarrow$ a lista com a $execution\_index_s$ de todos os processos
2: Dispõe buf em ordem decrescente
3: $index \leftarrow buf[f]$
4: **if** $index > commit\_index$ **then**
5: $commit\_index \leftarrow index$
6: **end if**

6. Avaliação

Esta seção avalia PRaft e o compara com os trabalhos relacionados. A avaliação de protocolos distribuídos normalmente é feita em termos de duas métricas: o número de passos de comunicação e o número de mensagens enviadas.

Avaliou-se neste trabalho o caso normal de processamento do protocolo, sem faltas nem troca de líder. PRaft segue o mesmo número de passos de comunicação que o Raft, mas tem apenas $n-1$ mensagens a mais. As mensagens contadas são $n-1$ *AppendEntry*s, $n-1$ mensagens de resposta ao *AppendEntry*, $n-1$ mensagens de término de processamento, e o $n-1$ *AppendEntry* seguinte que informa os seguidores do $commit\_index$ atualizado. As mensagens de votação não são contadas pois são feitas apenas uma vez por termo. As mensagens mais recorrentes são as três referidas, que somam $4(n-1)$ mensagens, onde n é o total de processos, e a complexidade é linear $O(n)$.

A tabela 1 mostra a comparação entre o PRaft e os algoritmos estudados. PRaft tem a menor complexidade de mensagens entre eles. O algoritmo de [Rodrigues et al. 1995] usa *view atomic multicast* [Schiper and Ricciardi 1993] que executa com $(n-1)^2$ mensagens, e [Wang et al. 2002] usa *general agreement framework* [Hurfin et al. 1999] que executa trocando $n+2n^2$ mensagens. Ambos possuem complexidade cúbica, devido aos serviços de consenso que usam. PritTO, de [Nakamura and Takizawa 1992] tem complexidade apenas quadrática, mas foi feito para sistemas síncronos. PRaft, tendo sido feito à semelhança do Raft [Ongaro and Ousterhout 2014], executa em tempo linear.

A Figura 2 mostra a comparação do desempenho de execução entre o Raft e o PRaft. Foram feitos testes com os dois algoritmos recebendo requisições de 20 clientes, que enviavam 100 requisições cada. Todos os clientes enviam requisições simultaneamente, e cada um só envia a próxima requisição depois que recebe a resposta da anterior. A cada requisição é associada uma prioridade aleatória entre 0 e 10, o que garante, por causa da regularidade da distribuição aleatória dos valores, que há um número aproximadamente igual de requisições em cada prioridade. O tempo de execução é fixo em 1 segundo.

A Figura 2 ilustra a latência da execução do Raft, notado com triângulos, e do PRaft, notado com cruzes. Como o Raft não distingue prioridades d as requisições, a média do tempo de execução é próxima de 18.91 segundos para todas as prioridades, com desvio padrão de 0.95 segundos. Já o PRaft discrimina a prioridade das requisições.

Na Figura 3 se podem ver os diagramas de barra da média de latência do PRaxos. No gráfico (a) está claro que requisições de prioridade 0 têm latência média muito maior que as demais. Isso acontece porque os clientes enviam as suas requisições sem pausa ao serviço, e se um cliente envia uma requisição de prioridade 0 no início dos testes, ele terá que esperar o processamento das requisições dos outros clientes antes de receber a resposta da sua. A média da latência para as requisições de prioridade 0 é de 256.28

Tabela 1. Comparação dos algoritmos relacionados

Algorítmo	Modelo de Tempo	# passos	# mensagens	Complex.
PritTO	síncrono	3	$(n-1)^2$	$O(n^2)$
Rodrigues et al.	detector de faltas	2	$(n-1)+n(n-1)^2$	$O(n^3)$
Wang et al.	detector de faltas	4	$(2n^3+7n^2+5n-2)/2$	$O(n^3)$
Raft	assíncrono	3	$3(n-1)$	$O(n)$
PRaft	*eventual synchronous*	3	$4(n-1)$	$O(n)$

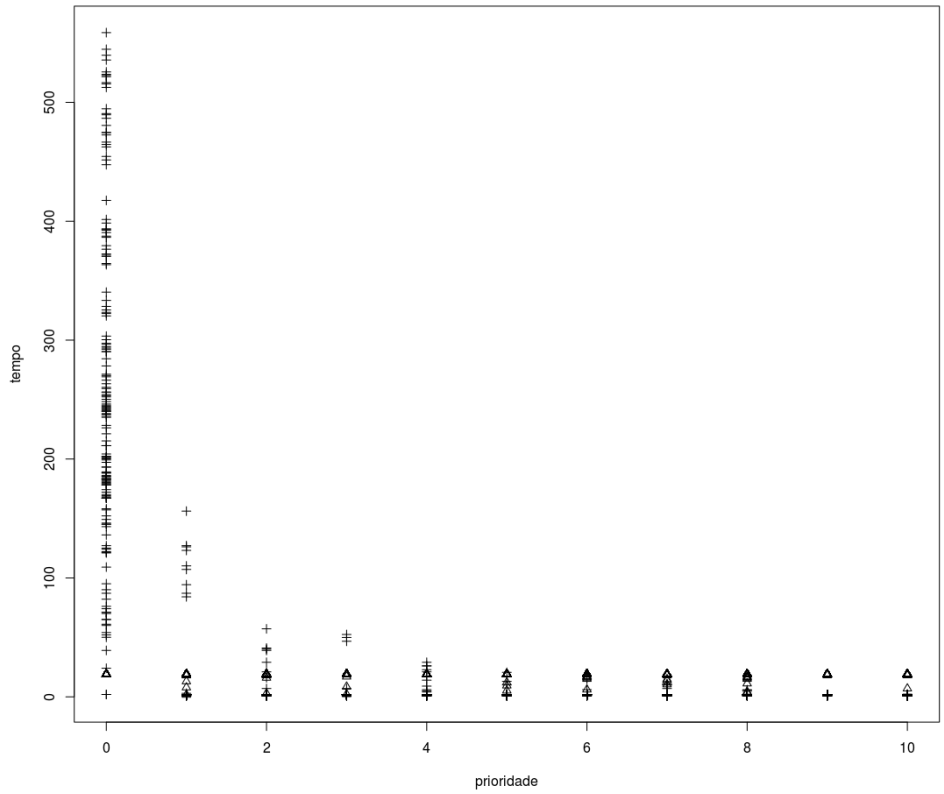
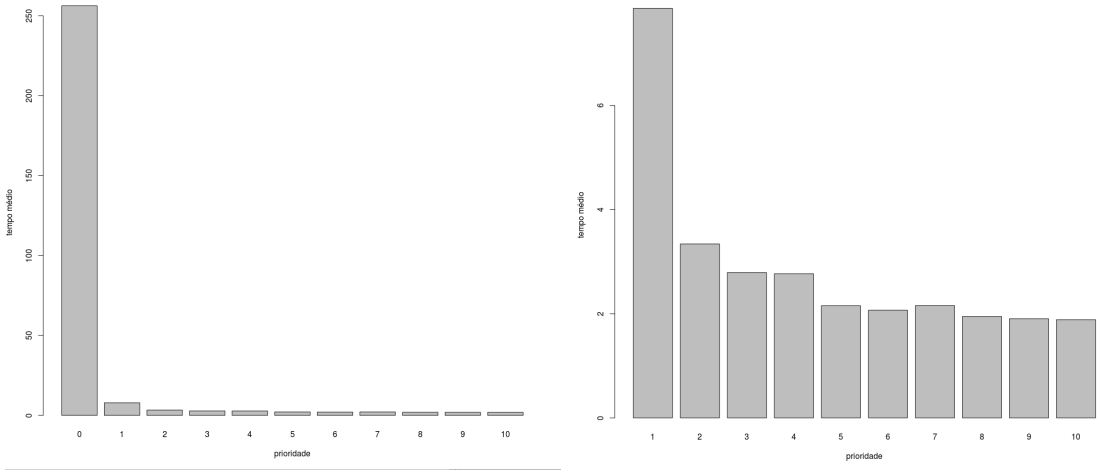


Figura 2. Desempenho comparativo entre a latência do Raft e do PRaft



(a) (b)

Figura 3. Diagramas de barra da média de latência do PRaft.

segundos, e alto desvio padrão de 134.63 segundos. O gráfico (b) mostra o diagrama de barras sem as requisições de prioridade 0 para facilitar a visualização. A latência das requisições diminui à medida em que aumenta a sua prioridade. A partir da prioridade 1, com latência média de 7.86 segundos, até as requisições de prioridade 10, que têm

latência média de 1.89 segundos, e desvio padrão de 0.31 segundos. Como se pode ver, a média da latência do Raft, 18.91 segundos, é maior do que a latência da requisição de prioridade 10 do PRaft, 1.89 segundos, isto é, dez vezes mais rápido. Isso demonstra que o PRaft é adequado para o processamento de requisições com prioridade.

7. Conclusão

Neste artigo foi proposto um algoritmo de consenso baseado em prioridade para a replicação de máquina de estado que não requer um serviço de consenso, e portanto funciona com menos mensagens que os algoritmos relacionados. Isso foi alcançado adaptando o algoritmo Raft para receber requisições e organizá-las segundo suas prioridades para que as de mais alta prioridade sejam executadas antes das de menor prioridades e evite inversão de prioridades. PRaft mantém as garantias de segurança (*safety*) e tolera $f = x/2$ faltas. As principais contribuições do PRaft são a execução antecipada das requisições, a ordem de prioridade, e a interrupção de execução. Esse protocolo pode ser utilizado na prática para suportar prioridades em serviços replicados como os de armazenamento em nuvem, serviços de arquivo em rede, *backup*s cooperativos e gerenciadores de banco de dados relacionais.

Referências

Aiyer, A. S., Alvisi, L., Clement, A., Dahlin, M., Martin, J., and Porth, C. (2005). BAR fault tolerance for cooperative services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*.

Bessani, A., Correia, M., Quaresma, B., André, F., and Sousa, P. (2011). DepSky: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of the 6th ACM SIGOPS/EuroSys European Systems Conference*, pages 31–46, Saltzburg, Austria. ACM, New York, NY.

Buyya, R., Yeo, C. S., and Venugopal, S. (2008). Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications*, pages 5–13.

Castro, M. and Liskov, B. (1999). Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 173–186.

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

Cristian, F. and Fetzer, C. (1998). The timed asynchronous system model. In *Proceedings of the 28th IEEE International Symposium on Fault-Tolerant Computing*, pages 140–149.

Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323.

Fetzer, C. and Cristian, F. (1995). On the possibility of consensus in asynchronous systems. In *Proceedings of the 1995 Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 86–91.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

Hadzilacos, V. and Toueg, S. (1993). Fault-tolerant broadcasts and related problems. In Mullender, S., editor, *Distributed Systems*, pages 97–145. ACM Press/Addison-Wesley.

Hurfin, M., Macedo, R. A., Raynal, M., and Tronel, F. (1999). A general framework to solve agreement problems. In *Proceeding of the 18th International Symposium on Reliable Distributed Systems*, pages 56–65.

Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. (2007). Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 45–58.

Lamport, L. (1998). The part-time parliament. *ACM Transactions Computer Systems*, 16(2):133–169.

Lamport, L. (2001). Paxos made simple. *ACM Sigact News*, 32(4):18–25.

Locke, C. D. (1986). *Best-effort Decision-making for Real-time Scheduling*. PhD thesis.

Luiz, A. F., Lung, L. C., and Correia, M. (2011). Byzantine fault-tolerant transaction processing for replicated databases. In *Proceedings of the 10th IEEE International Symposium on Network Computing and Applications*, pages 83–90.

Nakamura, A. and Takizawa, M. (1992). Priority-based total and semi-total ordering broadcast protocols. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 178–185.

Oki, B. M. and Liskov, B. (1988). Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 8–17.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference*, pages 305–319.

Rodrigues, L., Veríssimo, P., and Casimiro, A. (1995). Priority-based totally ordered multicast. In *3rd IFIP/IFAC Workshop on Algorithms and Architectures for Real-Time Control*.

Schiper, A. and Ricciardi, A. (1993). Virtually-synchronous communication based on a weak failure suspector. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, pages 534–543.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319.

Verissimo, P. and Rodrigues, L. (2001). *Distributed Systems for System Architects*. Kluwer Academic Publishers.

Wang, Y., Anceaume, E., Brasileiro, F., Greve, F., and Hurfin, M. (2002). Solving the group priority inversion problem in a timed asynchronous system. *IEEE Transactions on Computers*, 51(8):900–915.